PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 : H04N	A2	(11) International Publication Number: **WO 98/33315**
		(43) International Publication Date: 30 July 1998 (30.07.98)

(21) International Application Number: PCT/US97/23737

(22) International Filing Date: 31 December 1997 (31.12.97)

(30) Priority Data:

60/035,765	6 January 1997 (06.01.97)	US
60/034,549	7 January 1997 (07.01.97)	US

(71)(72) Applicants and Inventors: GEISLER, Wilson, S. [US/US]; 9801 Ravenwood Cove, Austin, TX 78750 (US). KORTUM, Philip, T. [US/US]; 6410 Walebridge Lane, Austin, TX 78739 (US).

(74) Agents: DENKO, J., Scott; Fulbright & Jaworski, L.L.P., Suite 2400, 600 Congress Avenue, Austin, TX 78701 (US) et al.

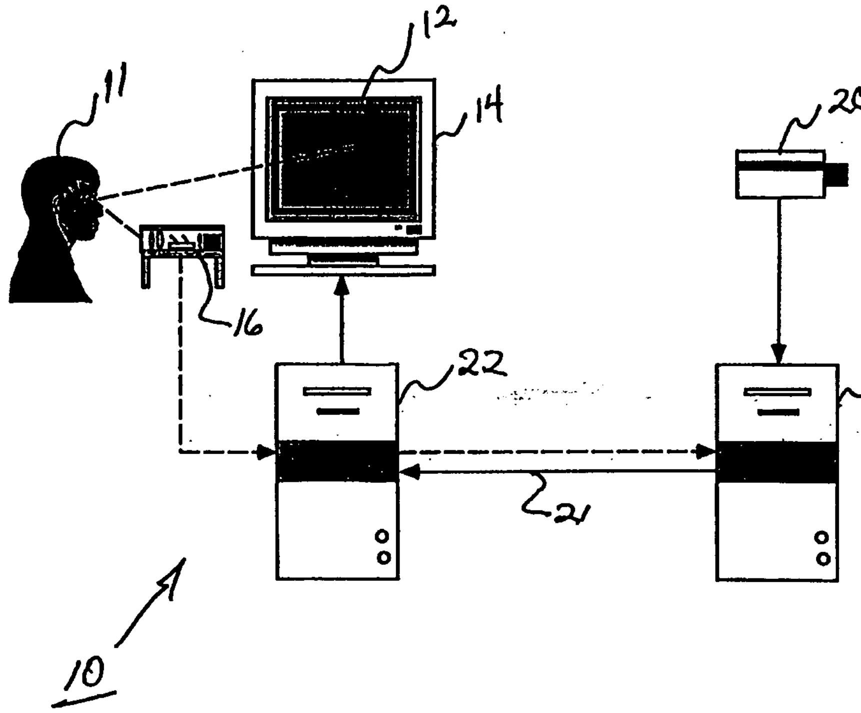
(81) Designated States: AL, AM, AT, AU, AZ, BB, BG, BR, BY, CA, CH, CN, CZ, DE, DK, EE, ES, FI, GB, GE, HU, IL, IS, JP, KE, KG, KP, KR, KZ, LK, LR, LS, LT, LU, LV, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, TJ, TM, TR, TT, UA, UG, US, UZ, VN, ARIPO patent (GH, GM, KE, LS, MW, SD, SZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, ML, MR, NE, SN, TD, TG).

Published
Without international search report and to be republished upon receipt of that report.

(54) Title: FOVEATED IMAGE CODING SYSTEM AND METHOD FOR IMAGE BANDWIDTH REDUCTION

(57) Abstract

A foveated imaging system, which can be implemented on a general purpose computer and greatly reduces the transmission bandwidth of images has been developed. This system has demonstrated that significant reductions in bandwidth can be achieved while still maintaining access to high detail at any point in an image. The system is implemented with conventional computer, display, and camera hardware. It utilizes novel algorithms for image coding and decoding that are superior both in degree of compression and in perceived image quality and is more flexible and adaptable to different bandwidth requirements and communications applications than previous systems. The system utilizes novel methods of incorporating human perceptual properties into the coding and decoding algorithms providing superior foveation. One version of the system includes a simple, inexpensive, parallel pipeline architecture, which enhances the capability for conventional and foveated data compression. Included are novel applications of foveated imaging in the transmission of pre-recorded video (without eye tracking), and in the use of alternate pointing devices for foveation.



Foveated Image Coding System and Method for Image Bandwidth Reduction

Related Applications:

This application claims priority under U.S. Title 35 § 119 from provisional application No. 60/034,549, filed January 7, 1997, in the United States, and provisional application no. 60/035,765, filed January 6, 1997, in the United States.

1.0. Rights in the Invention:

The United States government owns rights in the present invention pursuant to grant numbers AF94T004 and F49620-93-1-0307 from the Air Force Office of Sponsored Research.

1.1. Field of the Invention:

The present invention relates generally to the field of image data compression. More specifically, it relates to a foveated imaging system which can be implemented on a general purpose computer and which greatly reduces image transmission bandwidth requirements.

1.2. Description of the Related Art:

The human visual system provides detailed information only at the point of gaze, coding progressively less information farther from this point. This provides an efficient means for the visual system to perform its task with limited resources. Processing power is thus devoted to the area of interest and fewer neurons are required in the eye. Remarkably, we rarely notice the severely degraded resolution of our peripheral visual field; rather, we perceive the world as a single high resolution image that can be explored by moving our eyes to regions of interest. Imaging systems utilizing this fact, termed foveation, greatly reduce the amount of information required to transmit an image and therefore can be very useful for a number of applications including telemedicine, remote control of vehicles, teleconferencing, fast visual data base inspection, and transmission of pre-recorded video.

Recently, there has been substantial interest in such foveated displays. The U.S. Department of Defense has studied and used so-called "area-of-interest" (AOI) displays in flight simulators. These foveation schemes typically consist of only 2 or 3 resolution areas (rather than the superior continuous resolution degradation) and the central area of high resolution is often quite large, usually between 18° and 40° (see, for example, Howard, 1989; Warner, *et al.*, 1993). Other researchers have investigated continuous, variable-resolution methods using a log polar pixel configuration (Weiman, 1990; Juday and Fisher, 1989; Benderson *et al.*, 1992). The log polar configurations are particularly advantageous when rotation and zoom invariance are required, but their implementations have necessitated special purpose hardware for real-time operation.

Juday and Sampsell (U.S. Pat. No. 5,067,019) describe an apparatus which will encode a number of space-variant schemes. Fisher later refined the apparatus (U.S. Pat. No. 5,208,872). While otherwise effective, these systems are limited in that a specific hardware apparatus is necessary to perform the foveation operations. Further, these systems have utilized an Archimedes spiral to represent the falloff function applied to the image in their descriptions of the foveation method. While this may be computationally efficient for their specific hardware implementation, it does not as accurately represent the actual resolution falloff parameters of the human visual system, and does not provide the degree of flexibility in controlling the resolution function as the methods proposed here. Optimal compression and image quality are obtained by closely representing the actual resolution falloff parameters of the human visual system. The system of Weimans (U.S. Pat. No. 5,103,306), is more closely related to the apparatus described herein. However, implementation of the Weimans system requires specific hardware, and the foveation occurs in log polar coordinates, rather than Cartesian coordinates. Further, the algorithms employed during the compression and reconstruction of the foveated image create "cartoon-like" images (page 6, line 52, U.S. Pat. No. 5,103,306). Wallace, Benderson and Schwartz have also done work in the area of space-variant image processing (U.S. Pat. No. 5,175,617). They too, used a long polar mapping scheme, but because of the algorithms they utilized to perform the compression, the transmission rates were restricted to 4 frames per second, well below the rates required for near perceptually loss-less encoding.

2.0. Summary of the Invention:

The present invention overcomes the limitations in the prior art by providing a system that accomplished real-time foveated image compression and display, using general purpose computer processors, video hardware, and eye-tracking equipment. The foveated imaging system proposed here generates images that are nearly imperceptible from uncompressed images to the human observer. Further, the present invention makes use of a novel application of modified pyramid coding methods for creating foveated images. These coding methods provide higher quality images, at high compression rates, with little or no blocking or aliasing artifacts. The system can utilize one or more computer processors; the number of processors used depends upon the particular application and image-processing requirements.

2.1. Two-Processor FIS With Eye Tracking

In the inventors' Foveated Imaging System (FIS), eye movements are recorded in real time and are used to construct and display variable resolution images centered at the current point of gaze (fixation point). Typical operation of an exemplar FIS 10 proceeds as follows with reference to Fig. 1 for components and Fig. 2 for processes. First, the location of an observer 11 fixation point 12 on a display monitor 14 is measured with an eye tracking device 16 as depicted in process block 30 of Fig. 2. Second, the eye position coordinates are transmitted to a remote computer 18 as depicted in process block 32. Third, as represented by process block 34 of Fig. 2, remote computer 18 captures an image from camera 20. Fourth, the camera image is foveated (i.e., encoded and compressed so that the resolution of the image decreases from the point of fixation) as shown in process block 36. In other words, the degree of data compression increases with the distance from the point of fixation. Fifth, the encoded image is transmitted by communications channel 21 to a local computer 22 as shown in process block 38. Sixth, as depicted by process block 40, the received image is decoded and displayed on video monitor 14 such that the highest resolution region in the displayed image is centered at the fixation point 12. These six steps are repeated continuously in a closed loop. The system has been implemented in C^{++} for execution on standard PC compatible processors, including Intel

Pentium® and Pentium® Pro, but other general purpose processors could also be used.

FIS 10 can operate in any one of several different modes. The different modes correspond primarily to different methods of foveated image encoding and decoding. Mode 1 is the simplest and quickest, and hence is most appropriate for large images and/or high video frame rates. Mode 2 produces substantially better data compression, for the same image quality, but is somewhat slower. Mode 3 produces the greatest data compression, but is even slower, and hence is most appropriate for smaller image sizes and/or low frame rates. Mode 4 is also slower and produces somewhat poorer image quality than Modes 2 and 3; it is included because of its wide-spread use. As general-purpose processors become more powerful, there will be an increasing number of situations where the high compression modes will be appropriate.

Mode 1: Foveated Pixel Averaging

In Mode 1, variable size pixels (SuperPixels) are created by simple averaging (Kortum and Geisler, 1996). The sizes of the SuperPixels increase smoothly away from the point of fixation in a manner that matches the decline in resolution of the human visual system away from the point of fixation. The color of each SuperPixel is obtained by averaging the RGB (or gray level) values of the image pixels that fall within the boundary defined by the edges of the SuperPixel. The collection of integration boundaries defined by the edges of the SuperPixel is called the ResolutionGrid.

Fig. 3 illustrates one of the ResolutionGrids used in the system. ResolutionGrid 50 consists of a series of concentric rings of SuperPixels that increase in size away from the point of fixation, x_0, y_0. Because each of the SuperPixels is rectangular, SuperPixels can be represented with two pairs of coordinates (which define the summation bounds). This allows fast implementation of operations such as summation, clipping, and calculation of ResolutionGrid 50.

The size of the SuperPixels in the ith ring represented in Fig. 3 by reference character 52, is calculated according to the formula,

$$W_i = W_o \left(1 + \frac{\sqrt{(x_i - x_o)^2 + (y_i - y_o)^2}}{e_2}\right) \qquad (1)$$

wherein W_i is the size (in screen pixels) of the SuperPixels in ring 52, W_o is the size of the central foveal SuperPixel 54 (in screen pixels), x_0, y_o are the coordinates of the fixation point 56 (in degrees of visual angle), x_i,y_i are the coordinates of the nearest SuperPixel 58 in ring 52 (the i_{th} ring) (in degrees of visual angle), and e_2 is the half-resolution constant (in degrees of visual angle). This formula is based on available perceptual data, and is also consistent with anatomical measurements in the human retina and visual cortex (Wilson *et al.*, 1990; Geisler and Banks, 1995; Wassle *et al.*, 1992). For example, when e_2 is 2.5, the size of SuperPixel 54 is approximately proportional to the human resolution limit at each eccentricity. Thus, if W_0 is less than or equal to the foveal resolution limit then the foveated image should be indistinguishable from the original image (with proper fixation). If W_0 is greater than the foveal resolution limit, then the foveated image should be distinguishable from the original image. However, because W_0 is a proportionality constant in equation (1), the SuperPixel size will be above the resolution limit by a constant factor at all eccentricities. This should distribute any visible foveation artifacts uniformly across the display. Increasing e_2 increases the degree of foveation and hence the amount of compression.

Once ResolutionGrid 50 has been computed, a SuperPixel averaging subroutine averages the colors of each of the screen pixels that fall within each SuperPixel boundary, and assigns the resulting average value to that SuperPixel. SuperPixels at the edge of the display, and at the corners of the rings, are either clipped or merged in order to exactly cover the input image. SuperPixels of width 1 are excluded from the averaging subroutine.

Once the SuperPixels have been calculated, conventional secondary image compression algorithms are applied prior to transmission, decoding, and displaying. An example of a foveated image, without secondary compression is shown in Fig. 4. Fig. 4. is a 1024 x 1024 foveated image (Mode 1) with fixation at point 59 on the license plate of the Honda CRX. Because the pixel size is below the resolution limit of the visual system (for this size image), the resulting degradations due to foveation are imperceptible with appropriate fixation. The blockiness of the tree provides verification that this is a foveated image.

Mode 2: Foveated Laplacian Pyramid

Modes 2 and 3 make use of pyramid coding/decoding methods. Pyramid methods are widely used in non-foveated image compression (Burt & Adelson, 1983; Adelson, Simoncelli & Hingorani, 1987; Shapiro, 1993; Strang & Nguyen, 1996; Said & Pearlman, 1996). One new contribution of the Inventors' FIS is the modification of pyramid algorithms to include foveated image compression. Another contribution is the discovery that foveation not only increases the amount of image compression obtainable with pyramid methods, but substantially increases the speed of pyramid calculations. This makes foveated pyramid methods more feasible for real time video compression than non-foveated pyramid methods.

In Mode 2, a modified Laplacian pyramid algorithm (Burt & Adelson, 1983) is utilized. A simple example of the Foveated Laplacian Pyramid (FLP) algorithm is illustrated in Fig. 5. The pixels of an input image 60 are represented by the large grid in the upper left corner of Fig. 5. The coding steps are as follows for each level of the pyramid as shown in Fig. 5.

REDUCE. First, starting image 60 is filtered by convolution with a linear weighing function (kernal), and then down-sampled by a factor of 2 in each dimension, to obtain a smaller, lower-resolution image 62. In Fig. 5, the filter kernal is a 2 x 2 block, each element having a weight of 1/4; thus, this REDUCE operation is equivalent to computing 2 x 2 SuperPixels over the entire image (as in Mode 1). For example, b_{11} is the average of a_{11}, a_{12}, and a_{22}. Other kernals may also be used as later described in section 4.2.

EXPAND. Next, down-sampled (reduced) image 62 is interpolated (up-sampled) to obtain a larger, lower-resolution image 64. A great computational benefit of foveation is that it is usually only necessary to perform this EXPAND operation on a subset of the elements in the reduced image; i.e., on those elements within some distance of the fixation point, x_0, y_0. The elements over which the EXPAND operation is applied is determined from formulae derived from human perceptual data (see later) and from the current fixation position. Importantly, the percentage of elements over which the EXPAND operation needs to be applied is smallest for the largest reduced images (see later section 4.2).

DIFFERENCE. Next, expanded image 64 is subtracted from the starting image 60 to obtain a difference image 66. Again, there are computational savings over non-foveated pyramid methods since the DIFFERENCE operation is only applied to the image region that was expanded.

THRESHOLD. As indicated below, difference image 66 is thresholded, based upon formulae derived from human perceptual data (see 4.2). In the FIS, the threshold is a function of both the spatial frequency passband for the current pyramid level, and the distance to the fixation point. It is at this step that much of the image compression due to foveation is obtained. Again, there are also computational savings over non-foveated pyramid methods since the THRESHOLD operation is only applied to the image region that was expanded.

QUANTIZE. Next, the thresholded image is quantized to obtain additional data compression. Quantization is a form of lossy compression where the number of colors (or gray levels) used to represent image elements is reduced. In the FIS, the level of quantization is a function of both the spatial frequency passband for the current pyramid level, and the distance to the fixation point. In this step, further image compression due to foveation is obtained. Again, there are also computational savings over non-foveated pyramid methods since the QUANTIZE operation is only applied to the image region that was expanded.

LOSSLESS COMPRESSION. The final step in the coding process is standard lossless compression (e.g., Huffman or arithmetic coding, image differencing across frames). Again, there are also computational savings over non-foveated pyramid methods since LOSSLESS COMPRESSION is only applied to the image region that was expanded.

As shown in Fig. 5, all of the above operations are repeated for as many levels of the pyramid as desired. In the current FIS, all of the REDUCE operations are performed first. Computation of the thresholded, quantized and lossless-compressed difference images then begins with a smallest (lowest resolution) reduced image and proceeds toward the largest (highest resolution) reduced image.

TRANSMISSION. All of the coded difference images, and the final reduced image are transmitted to the local computer 22 (see Figs. 1 & 2), beginning with the final reduced image, and working backwards to the level-1 difference image. Each coded difference image is transmitted as soon as it is computed.

DECODING. Following transmission, the image data is decoded and displayed. The decoding proceedings in the reverse order of the coding. (1) Reverse the lossless compression. (2) Expand the quantize and thresholded data levels to the appropriate range. (3) Decode the pyramid to create the final transmitted image. During coding, it is only necessary to EXPAND those elements within some distance of the fixation point, x_0,y_o. However, for decoding it is necessary to apply EXPAND all of the elements at each level.

Mode 3: Foveated Wavelet Pyramid

Mode 3 utilizes a modified subband wavelet pyramid in place of the modified Laplacian pyramid. The processing steps in the Foveated Wavelet Pyramid are very similar to those of the Foveated Laplacian Pyramid; thus only the differences are described.

ANALYZE. In the wavelet pyramid the REDUCE, EXPAND and DIFFERENCE operations at each level of the coding pyramid are replaced by an ANALYZE operation which involves convolution with two or more linear kernals followed by downsampling. The most common 2D wavelet pyramids are based upon convolution with a low pass kernal, *l(x)*, and a high pass kernal *h(x)*, followed by down-sampling by a factor of 2 in each dimension (as in the Laplacian pyramid). The result, at each level of the pyramid, is four subband images: LL (low pass in both directions), HL (high pass in the x direction and low pass in the y direction), LH (low pass in the x direction and high pass in the y direction) and HH (high pass in both directions).

$$LL = 1(x)*(1(y)*i(x,y)) \cdot samp{\downarrow}(x,y) \qquad (2)$$

$$HL = h(x)*(1(y)*i(x,y)) \cdot samp{\downarrow}(x,y) \qquad (3)$$

$$LH = 1(x)*(h(y)*i(x,y)) \cdot samp{\downarrow}(x,y) \qquad (4)$$

$$HH = h(x)*(h(y)*i(x,y)) \cdot samp{\downarrow}(x,y) \qquad (5)$$

wherein i(x,y) represents the input image (or subimage), samp↓ (x,y) represents the sampling function (which sometimes must be different for each of the four equations), and * represents the operation of filtering (convolution). These four subimages together contain exactly the total number of elements in the input image.

The three high-pass subimages *(HL, LH, HH)* play the role of the difference image in the Laplacian pyramid. To them are applied the THRESHOLD, QUANTIZE and LOSSLESS COMPRESSION operations, prior to transmission. The strictly low-pass subimage (LL) plays the role of the reduced image in the Laplacian pyramid, it is used as the input image to the next level of the pyramid (e.g., it is processed by equations 2-5). The subimages for the first two levels of a wavelet pyramid are illustrated in Fig. 6.

The elements over which the high-pass convolutions are applied are determined from formulae derived human perceptual data (see later) and from the current fixation position. Because of the resolution properties of the human visual system, the percentage of elements over which the high-pass convolutions need to be applied is smallest for the largest subimages. This computational savings greatly increases the speed of the wavelet pyramid calculations, since only for the LL image (equation 2) must all the elements be processed.

SYNTHESIZE. Following transmission and initial decoding (undoing the LOSSLESS COMPRESSION, THRESHOLD and QUANTIZE operations) the subband images are synthesized into the final image which is then displayed. The SYNTHESIZE operation is similar to the final EXPAND and ADD operations of the Foveated Laplacian Pyramid. Specifically, the highest-level subimages are upsampled, filtered by a pair of low-pass and high-pass synthesis kernals and then added together. The resulting image and the subimages at the next level are then upsampled, filtered by a pair of low-pass and high-pass synthesis kernals and then added together. This process repeats until the full image is synthesized.

There are many possible choices for the low and high-pass kernals. Examples, are the 9-tap QMF filters of Adelson, Simoncelli & Hingorani (1987) or the 9/7-tap filters of Antonini, Barlaud & Daubechies (1992).

Mode 4: Foveated Block DCT

Mode 4 utilizes standard block DCT (Discrete Cosine Transform) coding/decoding rather than pixel averaging or pyramid coding. In block DCT the image is divided into non-overlapping blocks; a discrete cosine transform is then applied to each block (Stang and Nguyen, 1996). For example, in the JPEG standard the block sizes are 8x8 pixels. In Foveated Block DCT, the DCT coefficients are thresholded, quantized, lossless coded and transmitted in a fashion similar to that for the Foveated Pyramids. The DCT coefficients that need to be computed are determined from formulae derived from human perceptual data (see later) and from the current fixation position. Because of the resolution properties of the human visual system, the number of coefficients that needs to be computed decreases drastically away from the point of fixation. This computational savings greatly increases the speed of block DCT coding and decoding.

2.2. Eye Position Tracking

After initialization of parameters (and eye tracker calibration, if necessary), the system enters a loop in which the position of the eye is measured (see block 30 of Fig. 2). The measurement of eye position is done with a commercial eye tracking device 16. The current system uses a Forward Technologies Dual Purkinje Eyetracker, but almost any commercially available eye tracker would do. Eye position is measured and transmitted to the remote processor at approximately the frame rate of the display monitor. In Mode 1, the eye position is used to compute a new ResolutionGrid 50 centered on the point of fixation. In Modes 2 and 3, the eye position is used to set the region over which image elements are processed, and to set the threshold and quantization functions (see later).

Because of noise in the eye tracker and because of small fluctuations in eye position, it is usually desirable not to change the fixation point (x_o,y_o) used in the coding algorithm unless the change in measured eye position exceeds a threshold (e.g., 0.5 deg of visual angle). The rule used in the current system is:

$$\text{if } \sqrt{(x_0-x_m)^2 + (y_0-y_m)^2} < \delta \text{ then leave } (x_0,y_0) \text{ unchanged,}$$

$$\text{otherwise set } (x_0,y_0) \text{ equal to } (x_m,y_m) \qquad (6)$$

where (x_m,y_m) is the current measured eye position and δ is a threshold.

2.3 Four-Processor FIS with Eye Tracking

The process of foveation greatly reduces the amount of image data that needs further processing, and thus makes possible simple, inexpensive parallel pipeline architectures such as that in Fig. 7. The four processor system 70 illustrated in Fig. 7 consists of two remote processors 72 and 74 (e.g., two Pentium® computers) and two local processors 76 and 78. One of the remote processors 72 performs the initial foveation coding, while the other processor 74 performs the additional lossy and lossless compression, plus the transmission over the main communication line 21 to the local processors 76 and 78. One of the local processors 76 receives the remote data and performs the initial decoding into foveated image data, while the other processor 78 expands and displays the foveated image data. Because of the substantial data compression due to the foveation stage, simple, inexpensive, low bandwidth communication channels (see later) 80A and 80B are all that is needed between the two remote computers and between the two local computers. This system is capable of real time operation with larger image sizes and/or higher frame rates than the two-processor system 10. The only specialized piece of hardware required is a simple parallel communication card (or communications card of equivalent speed) in each computer.

This system 70 is particularly well adapted for Modes 2 and 3. As each difference image, or subband image, is computed in first the remote computer, it is transmitted to the other remote computer for thresholding, quantization, lossless coding and transmission. Similarly, as each difference or subband image is decoded by the first local computer, it is transmitted to the other local computer for expansion/synthesis and display.

2.4 FIS Without Eye Tracking

In general, foveated images appear low in quality unless fixation is at the proper position. Thus, for most imaging applications, eye tracking is an essential component of a useful foveated imaging system. There are, however, two exceptions.

One exception is when a mouse or some other pointing device can be used in place of the eyes. This is a simple modification of the system where the screen coordinates from the alternate pointing device are transmitted to the remote computer rather than the screen coordinates from the eye tracker. In all other respects, the system would be the same as described earlier. Alternate pointing devices could be useful in situations where eye tracking would be impossible or too expensive, yet not absolutely essential. Some examples would be surveillance systems or systems for inspecting/searching large data bases of images (e.g., satellite imagery). In these applications. pointing-device control of foveation might be adequate. In some cases, the remote control of vehicles, foot controls, touch pads, or pointing sticks would be adequate for controlling the foveation point.

The other exception is the presentation of pre-recorded video. In static images, observers typically fixate on a large number of points within the image, making observer-controlled foveation essential. However, in dynamic video situations, observers tend to fixate at certain critical points within the images. Thus, it might be possible to transmit foveated images based upon the most likely point of fixation, without observer-controlled foveation. Unfortunately, there are no adequate algorithms or models for predicting where observers will fixate in dynamic (or static) video sequences. However, for prerecorded video (e.g., movies), it would be possible to simply measure, in prior testing, where observers fixate the unfoveated video sequence. In other words, an eye tracker would be used in pre-testing to measure the point (or points) of fixation in every frame of the video, for each test observer. The fixation patterns of the test observers would then be used to create a foveated version of the pre-recorded video, using algorithms similar to those described in this document. The foveated version of the pre-recorded video would be transmitted to the user. The most obvious application of this type of foveated imaging system would be in transmission of pre-recorded video for high-definition digital television (HDTV). In this application, where very high perceptual quality is required, the amount of foveation would presumably be kept relatively small. Nonetheless, even a factor of two in additional compression (from foveation) would be of great significance.

2.5 Key Design Features of the Foveated Imaging System

The present invention contains many key design features. First, it can be implemented with conventional computer, display, and camera hardware, insuring low cost and easy improvements to the system. Most, if not all, previous systems that operate at useful video frame rates involve special purpose hardware that is costly and quickly becomes obsolete.

Second, the system utilizes novel algorithms (Foveated Laplacian Pyramids, Foveated Wavelet Pyramids and Foveated Block DCT) for image coding and decoding. These algorithms produce very high compression rates with high perceptual image quality. They are superior both in degree of compression and in perceived image quality to previous foveated imaging algorithms. Furthermore, Foveated-Laplacian-Pyramid, Foveated-Wavelet-Pyramid and Foveated-Block-DCT algorithms produce greater compression, and have faster execution times, than the non-foveated versions of these algorithms. The present system (unlike previous foveated imaging systems) makes practical real-time use of these highly-desirable coding algorithms on conventional, inexpensive, computer hardware (e.g., Pentium® class PCs).

Third, the system is designed to be highly flexible and adaptable to different situations. The overall display resolution, as well as the coding algorithm (the mode), is adjustable to accommodate different communication channel bandwidths and different image-size/frame-rate requirements. In addition, variable resolution, quantization and thresholding parameters are adjustable by the user to optimize subjective appearance or task performance.

Fourth, the system utilizes novel methods of incorporating human perceptual properties into the coding and decoding algorithms. Specifically, the system combines known variations in human contrast sensitivity, as function of spatial frequency *and* retinal eccentricity, in order to closely match the image compression to the image-coding properties of the human visual system. A close match insures that maximum compression is achieved with optimal perceptual quality. No previous system uses both smooth foveation based upon the decline in spatial resolution with eccentricity, and thresholding/quantization based upon the variation in contrast sensitivity with spatial frequency. The system's default variable-resolution algorithms, based upon quantitative (psychophysical) measurements of the variations in human contrast sensitivity with eccentricity and spatial frequency, provide superior foveation.

Fifth, the simple parallel pipeline architecture described in 2.3 and 4.5 enhances the capability for conventional data compression in conjunction with foveated compression. The represents an inexpensive method of increasing the image sizes and/or frame rates, and the level of image compression that can be obtained.

Sixth, the use of a foveated imaging system with alternate pointing devices, and in the transmission of pre-recorded video (without eye tracking), are novel applications.

The list of potential applications for the invention include:

- Teleconferencing (requires two foveated imaging systems)
- Remote control of vehicles
- Telemedicine
- Surveillance
- Fast visual data base inspection
- Transmission of pre-recorded video

3.0 Brief Description of the Drawings:

Fig. 1 depicts a preferred embodiment of a two-processor Foveated Imaging System.

Fig. 2 illustrates a general flow diagram of the operation of the two-processor Foveated Imaging System depicted in Fig. 1.

Fig. 3 illustrates a Foveated imaging system SuperPixel pattern arrangement called a ResolutionGrid.

Fig. 4 is a 1024 x 1024 foveated image (Mode 1).

Fig. 5 is an examplar of how a Foveated Laplacian Pyramid may be computed.

Fig. 6 is a depiction of two levels of a Foveated Wavelet Pyramid.

Fig. 7 depicts a preferred embodiment of a four-processor Foveated Imaging System.

Fig. 8 is a block diagram of PCI bus local communications card.

Fig. 9 is a general flow diagram for the C version of the Foveated Imaging System operating in Mode 1 (Foveated Pixel Averaging).

Fig. 10 is a general flow diagram for the C^{++} version of the Foveated Imaging System operating in Mode 1 (Foveated Pixel Averaging).

Fig. 11A graphically represents the process of creating the SuperPixel list in a preferred embodiment of the invention.

Fig. 11B graphically shows the process of averaging pixels from the input image.

4.0 Description of Preferred Embodiments

An important facet of the current invention is its ability to be run with conventional hardware, including computer, display, camera and eye-tracking equipment. In one embodiment of the invention, the computers may be any PC compatible Pentium® or Pentium® Pro with a PCI bus. Acceptable cameras include any NTSC (RS-170) camera (e.g., Sony XC-75) and many digital cameras (e.g. Dalsa CA-D2-512 digital camera, Dalsa, Waterloo, ONT, Canada). Conventional frame grabbers are also sufficient (e.g., MuTech MU-1000/MT-1300 (RS-170) or MU-1000/MV-1100 (digital), MuTech, Billerica, MA) as camera 20. Almost any commercially available eye-tracker 16 can be used in the system (e.g., Forward Technologies, San Marcos, TX; ISCAN Inc., Cambridge MA; Applied Science Laboratories, Bedford, MA; Express Eye, Renquishausen, Germany). To read the eye tracking signals, any A/D card (12 bits or better) and digital I/O card (8 bits or better), will suffice (e.g., Analogic HSDAS-16, Analogic, Wavefield, MA).

The only more specialized hardware would be a simple local communication interface card (able to support a transfer rate of 3 mbytes/sec) needed with the Four-Processor FIS system 70 (see 2.2 and Fig. 7). This communication card is not required for the Two-Processor

FIS system 10. A block diagram of one such card is shown in Fig. 16. The description of the blocks of Fig. 8 is as follows.

PCI Bus interface 100 — Implements capabilities required to attach a device to PCI bus 102 - including configuration registers, and a two quad-byte address space. The lower address represents the DATA in and out registers, the higher the CONTROL in and out registers. PCI Bus interface 100 also implements a 32 bit bi-directional data bus with a 1 bit address "bus" and read/write strobes for communicating with Data latches 104 and 105 and 3-state buffers 106.

Address and Read/Write decode logic 110 — Turns the PCI Bus interface 100 address bus and read write strobes into separate, address specific read and write strobes (read control, write control).

Data and Control output latches 104 and 105 hold Data written by PCI Bus interface 100, and make it available to the line drivers 112A and 112B for transmission to the other computer.

Data-in buffers 114 and Control-in 3-state buffers 116 pass data from bus receivers 117A and 117B (and from the other computer) to PCI Bus interface 100 in response to read strobe signals from the PCI Bus interface 100 (through the Address and Read/Write decode logic).

DAV logic block 118, together with the ACK logic block of the other computer implement handshaking between computers. DAV Logic 118 generates a DATA AVAILABLE (DAV) signal to notify the receiving computer that data has been placed on the data out lines and should be read. The DATA AVAILABLE signal remains asserted until the receiving computer asserts the DATA ACKNOWLEDGE (ACK) signal line, signifying that the transmitted data has been read. The DAV signal is also connected to Control-In 3-state buffers 116 to allow the program controlling the transmission of data to determine when a previously transmitted data has been safely received.

ACK Logic 120 implements the receiving end of the DAV/ACK handshaking. Receipt of a DAV signal notifies the controlling program, through the Control-In section, that data is available and must be read. When the controlling program reads the Data-In register, ACK Logic 120 asserts the ACK signal. When the other computer sees the ACK signal, it de-asserts it's DAV signal, which causes ACK logic to de-assert its ACK signal. The system is then ready for the next transmission.

Line drivers and receivers will depend on transmission distance requirements. For short distances and lower data rates, none will be needed. For longer distances and higher data rates, more sophisticated drivers and receivers can be used.

4.1 Foveated Pixel Averaging

One version of the system, with Foveated Pixel Averaging, has been implemented in C (Portland Group PGCC) for execution on an ALACRON 1860 (Alacron Inc., Nashua, NH) processor, but better performance can be obtained on a Pentium® Pro processor. Fig. 9 illustrates the general function of the system. Upon system initialization depicted by block 200, the user is queried for a number of required parameters (pixel dimensions, image size, image distance, half resolution constant, eye movement update threshold, etc.). Using these parameters, a space variant arrangement of SuperPixels (referred to as the Resolution Grid) is then calculated and stored for display as shown by process block 202 (a SuperPixel is a collection of screen pixels that have been assigned the same gray level value). Next, an eye position calibration is conducted shown by process block 204. The system then enters a closed loop mode, in which the current eye position is determined and compared with the last known eye position as depicted in process block 206. If the eye has not moved more than some amount specified during initialization (decision block 208), then pixel averaging (based on the Resolution Grid) is executed as shown in block 210. However, if the eye has moved more than this threshold amount then the coordinates of the ResolutionGrid are offset by the magnitude of the eye movement before pixel averaging as depicted by process block 209. Finally, the resulting SuperPixels are sent to the display device as shown in block 212, and eye position is checked again.

Only the closed loop portion of the program is required to run in real time. Initialization, calibration and calculation of the space-variant resolution grid take place prior to image display. However, because of the simplicity of the resolution grid structure (which is described in greater detail below), it can also be re-calculated in real time, if desired.

The system has also been implemented in C++ (Watcom C/C++) for execution on an Intel Pentium® processor. This implementation works similarly to the C version: parameters are entered by the user and the system is initialized, and the system then enters a closed loop in which eye movement data is collected. Within the closed loop, the image is foveated and displayed based upon the movement of the eye.

One difference between the C implementation and the C++ implementation is that whenever the eye moves beyond the eye movement threshold, the C++ implementation recomputes the entire ResolutionGrid based upon the new eye position. In the C implementation, the ResolutionGrid must be 4 times greater than the image in order to facilitate offsetting the grid relative to the fixation point. Each SuperPixel in the ResolutionGrid must then be examined to determine whether or not it lies within the image boundary. The Superpixels that do not lie within the image boundary are then clipped. In the C++ implementation, since the Resolution Grid is recalculated with each eye movement (that exceeds the eye movement threshold), the grid needs only be as large as the image, and, therefore, no clipping is required. In these particular implementations, recomputing the ResolutionGrid and not clipping SuperPixels has shown a slight performance benefit over computing a single ResolutionGrid and offsetting it relative to the fixation point.

Fig. 10 illustrates a flowchart description of Foveated Pixel Averaging and the processes of creating the Superpixel list, averaging pixels from the input image, and displaying them on the display are summarized in Figs. 11A, 11B, and 11C, respectively.

Block 214 in the flow chart of Fig. 10, shows the initialization phase. During the initialization phase parameters are entered and set as described above.

After initialization, the current measured eye position (x_m, y_m), is obtained as shown by blocks 215 and 216. The eye position's distance from the systems's fixation point, $(x_0 y_0)$, is then compared to a threshold, δ in decision block 218. If $\sqrt{(x_0-x_m)^2 + (y_0-y_m)^2} < \delta$ then (x_0, y_0) remains unchanged, and the "No" branch of the flowchart is taken. Otherwise (x_0, y_0) is set equal to (x_m, y_m), and the "Yes" branch of the flowchart is followed in order to create a new list of SuperPixels as depicted by block 220.

When a new list of SuperPixels is created, the system's fixation point and the parameters given during initialization are used to determine each SuperPixel's properties. Each SuperPixel in the list specifies a size and starting coordinate relative to the viewed image. The final list collectively describes a non-overlapping grid of pixel regions that cover the entire image. Creation of the SuperPixel list is graphically shown in Fig. 11A.

The next step in the process is to average pixels in the input image in order to get an average color or grey scale value for each SuperPixel as shown in block 222. Averaging of grey scale values is done by simply summing pixels over the superpixel's pixel region and dividing by the number of pixels in the superpixel. Averaging of unmapped, or true-color values is done the same as for grey scales, except the individual red, green, and blue intensities are each averaged. Averaging of color mapped images (images whose pixel values correspond to indices in a map of color values) is achieved by first converting each superpixel's RGB (tristimulus) components to YUV (color sum/difference) components. These YUV components are then averaged, and the resulting average YUV components are used to find the closest matching color value in the image's color map. The averaging process is graphically shown in Fig. 11B.

The final step is to write the Superpixels to the display in process block 224. The simplest way to do this is to write each pixel as a block whose pixels are all the same value as the Superpixel's averaged color or grey scale value. A better way is to interpolate pixel values in order to create a smooth image that does not contain block edges. In order to do this, each Superpixel's grey scale or color value is written to the display buffer in the center of that Superpixel's pixel region. Then, bilinear interpolation is used to smooth the image. Specifically, pixels are linearly interpolated in the vertical direction to create smooth color changes between all vertically adjacent superpixel centers. Pixels are then linearly interpolated in the horizontal direction to create smooth color changes between the vertically interpolated pixels. Constructing an image is graphically shown in Fig. 11C.

4.2 Foveated Laplacian Pyramid

The Foveated Laplacian Pyramid system has been implemented in C++ (Watcom C/C++) on an Intel Pentium® processor.

Foveated Laplacian Pyramids work similarly to Laplacian Pyramids, however, during the expand operation of the Foveated Laplacian Pyramid, only the region within the human contrast threshold needs to be Expanded and Differenced during the encoding process. This means that during the decoding process, only in this same region will differenced images need to be added back to obtain the original image. This is illustrated in Fig. 12 and Fig. 13.

REDUCE. In the 2 x 2 version, the REDUCE operation works as follows. Each 2 x 2 block in Level *l* is averaged to obtain a pixel in level *l+1*. For example, as shown in Fig. 5 and Fig. 14,

$$b_{11} = \frac{a_{11}+a_{12}+a_{21}+a_{22}}{4} \tag{7}$$

In the 3 x 3 version shown in Fig. 15, the REDUCE operation works as follows. Each 3 x 3 block in Level *l* is averaged using the following weighing function to obtain a pixel in Level *l*+1:

$$W = \begin{matrix} [1/16 & 1/8 & 1/16] \\ [1/8 & 1/4 & 1/8] \\ [1/16 & 1/8 & 1/16] \end{matrix} \tag{8}$$

For example,

$$b_{11} = \frac{a_{11}+2a_{12}+a_{13}}{16} + \frac{a_{21}+2a_{22}+a_{23}}{8} + \frac{a_{31}+2a_{32}+a_{33}}{16} \tag{9}$$

$$b_{12} = \frac{a_{13}+2a_{14}+a_{15}}{16} + \frac{a_{23}+2a_{24}+a_{25}}{8} + \frac{a_{33}+2a_{34}+a_{35}}{16} \tag{10}$$

EXPAND. The EXPAND operation for the 2 x 2 version works as follows and as shown in Fig. 16. First note that the solid dots at the grid intersection, at the top of Fig. 5, show the appropriate spatial location of the down-sampled pixels representing the 2 x 2 block averages. Thus, for example, b_{11} in the Level 2 grid is the local average color (e.g., gray level) at the position indicated by the solid dot in the Level 1 grid. This is illustrated more fully in Fig. 18, where the Level 2 pixels have been overlaid on the Level 1 grid.

The purpose of the EXPAND operation is to interpolate/expand the reduced image to its size prior to applying the REDUCE operation. For each pixel location in the expanded image, the nearest three pixels in the reduced image are found. For example, the three pixels in the reduced image that are nearest to a_{22} are b_{11}, b_{12} and b_{21}.

The three nearest pixels define a plane in color (or gray scale) space. The interpolated value is obtained by evaluating the color plane at the location of the pixel in the expanded image. The algebra of this calculation is to first find the average of the two pixels defining each of the perpendicular sides of the triangle and then to find the average of those two averages. For example,

$$\bar{a}_{22}=\frac{\frac{b_{11}+b_{12}}{2}+\frac{b_{11}+b_{21}}{2}}{2}=\frac{b_{11}}{2}+\frac{b_{12}+b_{21}}{4} \qquad (11)$$

Similarly,

$$\bar{a}_{23}=\frac{b_{12}}{2}+\frac{b_{11}+b_{22}}{4}, \quad \bar{a}_{33}=\frac{b_{22}}{2}+\frac{b_{12}+b_{21}}{4}, \quad \bar{a}_{32}=\frac{b_{21}}{2}+\frac{b_{11}+b_{22}}{4} \qquad (12)$$

The EXPAND operation for the 3 x 3 version is simpler. The sold dots in Fig. 19 show the appropriate spatial locations of the down-sampled pixels representing the 3 x 3 averages. The interpolated values are obtained by bi-linear interpolation. The interpolated pixels between the level $l+1$ pixels are obtained by simple averaging. For example,

$$\bar{a}_{43}=\frac{b_{21}+b_{22}}{2} \text{ and } \bar{a}_{34}=\frac{b_{12}+b_{22}}{2} \qquad (13)$$

The remaining interpolated pixels are obtained by averaging the initially interpolated pixels. For example,

$$\bar{a}_{33}=\frac{\bar{a}_{23}+\bar{a}_{43}}{2} \qquad (14)$$

(See also Fig. 18).

DIFFERENCE. The DIFFERENCE operation is to simply subtract the expanded image from the starting image. For example, for each location *ij* in the Level 1 (Fig. 5),

$$a_{ij} = a_{ij} - \bar{a}_{ij} \tag{15}$$

THRESHOLD. The color (or gray level) of each pixel, in each difference image, is thresholded in a fashion that varies with level in the pyramid and with distance from the point of fixation (eccentricity). Different threshold functions can be selected based upon the application. Here is described a particular threshold function for gray scale images. To maximize perceptual image quality the inventors have implemented a threshold function which is consistent with measurements of contrast sensitivity in the human visual system as function of spatial frequency and retinal eccentricity (Wilson et al. 1990; Geisler and Banks, 1995).

Let x_0,y_0 be the point of fixation in the image expressed in degrees of visual angle from the center of the image. If a pixel is located at position *x, y*, then the eccentricity, *e*, is given by:

$$e = \sqrt{(x-x_0)^2 + (y-y_0)^2} \tag{16}$$

The threshold function is given by the following equation:

$$G_T(e,1) = G_{max} \,.\, C_T(e,1) \tag{17}$$

where G_{max} is the maximum possible absolute value in the difference image (e.g., 128 for 8-bit gray level), and $C_T(e,1)$ is the human contrast threshold for the eccentricity, *e*, and for the spatial frequency passband associated with Level *l*. By definition the maximum contrast threshold is 1.0.

The purpose of the threshold is to eliminate all difference pixels which are below human contrast threshold. If *G* is the value of the pixel at location *x,y* in the difference image for level *l* then the post-threshold value is given by,

$$G' = \begin{pmatrix} G-G_T(e,1) \text{ if } G \geq G_T(e,1) \\ G+G_T(e,1) \text{ if } G \leq -G_T(e,1) \\ 0 \text{ otherwise} \end{pmatrix} \tag{18}$$

Notice that when contrast threshold for a pixel reaches a value of 1.0 then difference-image value cannot exceed the threshold, and hence the pixel need not be transmitted. Additional compression is obtained because equation (18) reduces the range of difference values.

In the current version of the algorithm we use the following formula for human contrast threshold although other similar formulas could be used:

$$C_T(e,1) + \min\left(C_0 \exp\left(\alpha\left(\frac{u_1\ (e+e_2)}{e_2}\right)^2\right),1\right) \tag{19}$$

where C_0 is the minimum contrast threshold (across all spatial frequencies and eccentricities), u_l is the dominant spatial frequency of the passband of Level l, e_2 is the eccentricity at which human spatial resolution falls to one-half the resolution at zero eccentricity and α is a constant. The inventors have found that this formula provides a reasonable approximation to the falloff in the contrast sensitivity of the human visual system as a function of spatial frequency u_l and eccentricity e.

Setting $C_T(e,1)$ to 1.0 and solving for e yields a formula which gives the eccentricity, e_{max}, beyond which it is unnecessary to transmit pixel information *and* unnecessary to perform the EXPAND operation:

$$e_{max} = \frac{e_2}{u_1}\sqrt{\frac{InC_0}{-\alpha}} - e_2 \tag{20}$$

For low levels (e.g., Level 1) the dominant spatial frequency, u_1, is high and hence e_{max} is small. Thus, the greatest compression and computational savings are obtained for the largest size difference image.

QUANTIZE. Further compression is obtained by quantization. Many image compression algorithms make use of the fact that humans are able to resolve fewer levels of gray in higher spatial frequency bands than in low frequency bands. We also allow the possibility of having the level quantization vary with retinal eccentricity. If G' is the thresholded value of the difference-image pixel at location x,y, then the value after quantization is given by,

$$G'' = NINT(Q(e,1).G') \tag{21}$$

where NINT is the "nearest integer" function, and Q(e,1) is the quantization function:

$$0 \leq Q(e,1) \leq 1.0 \qquad (22)$$

LOSSLESS COMPRESSION. The final step before transmission is lossless compression. (1) If the point of foveation has not changed then, for each level of the pyramid, only the difference between the current compressed image and the previous image is transmitted. If the point of foveation has changed then frame differencing is not used. (2) Huffman or arithmetic coding is applied to the final images for each level of the pyramid.

DECODING. Following transmission, the image data is decoded and displayed. The decoding proceeds in the reverse order of the coding. (1) Reverse Huffman or arithmetic coding. (2) Add the difference image to the previous image (if point of fixation was the same). (3) Divide G" by Q(e,1) to expand the gray levels to G' values. (4) Add $G_T(e,1)$ to G' if G' > 0, subtract $G_T(e,1)$ from G' if G'<, 0 to expand gray levels to G values. (5) Decode the pyramid to create the final transmitted image. For example, decoding of the coded video image data of Fig. 5 would proceed as follows: EXPAND the Level 4 image and ADD the result to the Level 3 difference image to obtain the Level 3 image, EXPAND the Level 3 image and ADD the result to the Level 2 difference image to obtain the Level 2 image, EXPAND the Level 2 image and ADD the result to the Level 1 difference image to obtain the Level 1 image.

Note that for decoding it is necessary to apply EXPAND to the entire image at each level (se Fig. 13). Whereas, during coding it is only necessary to apply EXPAND within the region defined by e_{max}(see Fig. 12).

4.3 Foveated Wavelet Pyramid

The Foveated Wavelet Pyramid may be implemented in C^{++} (Watcom C/C^{++}) on an Intel Pentium® processor. The computations can be of the form typical in the literature, with the addition of the modifications for foveated imaging described above. Specifically, the THRESHOLD and QUANTIZE operations will be applied to each of the subimages. The only difference is that l in equations (17)-(22) will now represent a subimage of the wavelet pyramid, and u_l will represent the dominant spatial frequency of the passband associated with that subimage. Equation (20) will be utilized to determine the range of elements for which the high-pass filtering (h) needs to be computed.

4.4 Foveated Block DCT

The Foveated Block DCT may be implemented in C++ (Watcom C/C++) on an Intel Pentium® processor. The computations will be of the form typical in the literature, with the addition of the modifications for foveated imaging described in 2.1. Specifically, the THRESHOLD and QUANTIZE operations will be applied to each of DCT components (coefficients) computed in each block. The only difference is that l in equations (17)-(22) will now represent a particular DCT component (i.e., basis function) in a particular block, u_l will represent the dominant spatial frequency of the component, and e will represent the eccentricity of the center of the particular block. Equation (20) will be utilized to determine those blocks for which each DCT component must be computed.

4.5 FIS Perceptual Evaluation

The current invention (running under Mode 1) has been evaluated for perceptual performance using a conventional 21 inch video display monitor. The images were 256 x 256, 8 bit gray scale images, having a 20° field of view, displayed with eye movement update rates of 30 Hz, at a frame rate of 60 Hz, and with a half-resolution constant (e_2) of 2.5°. The images were selected to test the perception of a number of probable image types: a letter chart (evaluation of visual clarity in a reading task), a face (evaluation of video telecommunication systems), and a natural environment scene (evaluation of cluttered, high detail images). User reports of the subjective quality of the displays were used in the evaluation. All subjects have reported smooth, accurate tracking and excellent overall functioning of the foveating algorithms. Upon steady fixation, most subjects noted that they were aware of the reduced resolution in the peripheral visual field, but that the effect was minimal and had only a small impact on the perceived quality of the image.

In a more quantitative study, the inventors measured visual search performance (in Mode 1) for a 1.0° x 0.6° target in natural scenes, under three different conditions: (1) foveated imaging of 20° x 20° images, (2) windowed viewing (4.9° x 4.9°) with the number of pixels equal to that of the foveated image, and (3) constant resolution viewing (20° x 20°) with the number of pixels equal to that of the foveated image. The results showed that the search speed and accuracy was much greater for the foveated imaging. These results illustrate the potential value of the foveated imaging system in important real-world applications where transmission bandwidth is limited.

4.6 Computational Performance and Bandwidth Reduction

Use of the Foveated Imaging System has demonstrated that significant bandwidth reduction can be achieved, while still maintaining access to high detail at any point of a given image. In Mode 1, using 19 x 22.5 cm. images, at a viewing distance of 30 cm, with a half resolution constant (e_2) of 2.5° and a foveal SuperPixel size of 1 pixel, the inventors are able to achieve bandwidth reductions of 97% (a factor of 32) for 644 x 768 images, on a 200 mHz Pentium® computer. For these systems parameters and images, the eye-movement update rate for thee display exceeded 12 Hz for 24-bit color images, and exceeded 17 Hz for 8-bit gray scale images (the refresh rate of the display was 60 Hz). These bandwidth reductions are independent of the particular images and are obtained without any form of secondary compression. A computer coding used in a preferred embodiment of the present invention is included with this application as an Appendix I.

Secondary image compression schemes, such as thresholding, quantization, and image-frame differencing, can be used in conjunction with the foveated imaging system. The effects of such secondary compression are essentially multiplicative; for example, a factor of 20 in secondary compression would yield an overall compression factor of 640.

Non-foveated Laplacian and wavelet pyramids often produce quite acceptable 8-bit gray images with bit rates of 0.1—0.5 bits/pixel (Burt & Adelson, 1984; Said & Pearlman, 1996). The Foveated Laplacian and Foveated Wavelet Pyramids should produce quite acceptable foveated images at bit rates many times lower.

It will be obvious to those having skill in the art that many changes may be made in the above-described details of a preferred embodiment of the present invention without departing from the underlying principles thereof. The scope of the present invention should, therefore, be determined only by the following claims.

APPENDIX 1

```
CURSOR.CXX:
#include "cursor.h"

void Cursor::UpdateX (int x)
{
    _x += x;

    if (_x > (int) (_max_x - _width / 2 - 1))
        _x = _max_x - _width / 2 - 1;
    else if (_x < (int) _width / 2)
        _x = _width / 2;
}

void Cursor::UpdateY (int y)
{
    _y += y;

    if (_y > (int) (_max_y - _width / 2 - 1))
        _y = _max_y - _width / 2 - 1;
    else if (_y < (int) _width / 2)
        _y = _width / 2;
}

// Turn off the following two warnings:
// W604: must look ahead to determine whether construct
//           is a declaration/type or an expression
// W594: construct resolved as a declaration/type

#pragma warning 604 9
#pragma warning 594 9

void Cursor::Draw (void (*LineDraw) (unsigned x1, unsigned y1, unsigned x2, unsigned y2, unsigned long color))
{
    // +

    LineDraw (_x + _offset_x - _width / 2,
        _y + _offset_y,
        _x + _offset_x - _width / 4,
        _y + _offset_y,
        _color1);
    LineDraw (_x + _offset_x - _width / 2,
        _y + _offset_y + 1,
        _x + _offset_x - _width / 4,
        _y + _offset_y + 1,
        _color1);

    LineDraw (_x + _offset_x + _width / 2,
        _y + _offset_y,
        _x + _offset_x + _width / 4,
        _y + _offset_y,
        _color1);
    LineDraw (_x + _offset_x + _width / 2,
        _y + _offset_y + 1,
        _x + _offset_x + _width / 4,
        _y + _offset_y + 1,
        _color1);

    LineDraw (_x + _offset_x,
        _y + _offset_y - _width / 2,
        _x + _offset_x,
        _y + _offset_y - _width / 4 + 1,
        _color1);
    LineDraw (_x + _offset_x + 1,
        _y + _offset_y - _width / 2,
        _x + _offset_x + 1,
        _y + _offset_y - _width / 4 + 1,
        _color1);

    LineDraw (_x + _offset_x,
        _y + _offset_y + _width / 2,
        _x + _offset_x,
        _y + _offset_y + _width / 4,
        _color1);
    LineDraw (_x + _offset_x + 1,
        _y + _offset_y + _width / 2,
        _x + _offset_x + 1,
        _y + _offset_y + _width / 4,
        _color1);

    // x

    LineDraw (_x + _offset_x - _width / 2,
        _y + _offset_y - _width / 2,
        _x + _offset_x - _width / 4,
        _y + _offset_y - _width / 4,
        _color2);
    LineDraw (_x + _offset_x + _width / 2,
        _y + _offset_y + _width / 2,
        _x + _offset_x + _width / 4,
        _y + _offset_y + _width / 4,
        _color2);
    LineDraw (_x + _offset_x + _width / 2,
        _y + _offset_y - _width / 2,
        _x + _offset_x + _width / 4,
        _y + _offset_y - _width / 4,
        _color2);
    LineDraw (_x + _offset_x - _width / 2,
        _y + _offset_y + _width / 2,
        _x + _offset_x - _width / 4,
        _y + _offset_y + _width / 4,
        _color2);
}

FOVEATOR.CXX:
#include <assert.h>
#include <math.h>
```

```
#include "foveator.h"

Foveator::Foveator ()
{
    _half_resolution = 0.0;
    _distance = 0;
    _pixels_per_cm = 0;
    _w0 = 0;
    _x_resolution = 0;
    _y_resolution = 0;
    _max_super_pixels = 0;
    _super_pixel_index = 0;
    _super_pixels = 0;
}

Foveator::~Foveator ()
{
    if (_super_pixels)
        delete[] _super_pixels;
}

void Foveator::SetScreenResolution (unsigned x_resolution, unsigned y_resolution)
{
    _x_resolution = x_resolution;
    _y_resolution = y_resolution;

    if (_super_pixels)
        delete [] _super_pixels;

    // The number of superpixels allocated is based upon the image size.

    _max_super_pixels = _x_resolution * _y_resolution;
    _super_pixels = new SuperPixel [_max_super_pixels];

    if (!_super_pixels)
        throw "Foveator::AllocateSuperPixels(): memory allocation error";
};

SuperPixel *Foveator::Foveate (unsigned center_x, unsigned center_y, unsigned &super_pixel_count)
{
    if (center_x > _x_resolution)
        throw "Foveator::Foveate(): X center value is out of range";

    if (center_y > _y_resolution)
        throw "Foveator::Foveate(): Y center value is out of range";

    if (!_x_resolution || !_y_resolution)
        throw "Foveator::Foveate(): a non-zero X and Y resolution must be specified";

    if (!_distance)
        throw "Foveator::Foveate(): a non-zero value for distance be specified";

        const double PI = 2 * asin (1.0);
    unsigned max_resolution = (_x_resolution < _y_resolution) ? _y_resolution : _x_resolution;

    // Compute the super pixels

    _super_pixel_index = 0;
    unsigned pixel_size;

    float half_res_pixels = _half_resolution * tan (2 * PI / 360) *
              _distance * _pixels_per_cm;

    for (unsigned x = 0; x < max_resolution; x += pixel_size)
    {
        pixel_size = (unsigned) ((_w0) * (1.0 + x / half_res_pixels) + 0.5);

        if (pixel_size < 1)
            pixel_size = 1;

        // // Round to nearest power of 2
        //
        // unsigned log_2 = (unsigned) (log (pixel_size) / log (2.0));
        //
        // pixel_size = 1 << log_2;

        unsigned pixel_height;

        for (unsigned y = 0; y <= x; y += pixel_height)
        {
            pixel_height = pixel_size;

            // If the next superpixel is going to be truncated,
            // decide whether or not we should connect it to this
            // super pixel.

            if (y < x && y + pixel_height + pixel_height / 2 > x)
                pixel_height = x - y;
            else if (y < x && y + pixel_height > x)
                pixel_height = x - y;

            AddSuperPixels (center_x, center_y, x, y, pixel_size, pixel_height);
        }
    }

    super_pixel_count = _super_pixel_index;

    return_super_pixels;
}

void Foveator::AddSuperPixels (unsigned center_x, unsigned center_y,
                               unsigned x, unsigned y,
                               unsigned width, unsigned height)
{
    int x1 = x;
    int y1 = y;
    int x2 = x + width;
    int y2 = y + height;

    // Rotate about center counter clockwise starting at 3:00

    AddSuperPixel (center_x + x1, center_y + y1, width, height);
    AddSuperPixel (center_x + y1, center_y - x2, height, width);
    AddSuperPixel (center_x - x2, center_y - y2, width, height);
    AddSuperPixel (center_x - y2, center_y + x1, height, width);

    // Only add superpixels that are on the diagonal one time

    if (x == y)
        return;

    // Mirror about the diagonal

    x1 = y;
    y1 = x;
    x2 = y + height;
    y2 = x + width;

    // Rotate as before, switching width and height

    AddSuperPixel (center_x + x1, center_y + y1, height, width);
    AddSuperPixel (center_x + y1, center_y - x2, width, height);
    AddSuperPixel (center_x - x2, center_y - y2, height, width);
    AddSuperPixel (center_x - y2, center_y + x1, width, height);
}

void Foveator::AddSuperPixel (int x, int y, unsigned width, unsigned height)
{
    int left = x;
    int bottom = y;
    int right = x + width - 1;
    int top = y + height - 1;

    // Check to see if it is within screen limits

    if (top < 0 | right < 0)
        return;

    if (left > (int) (_x_resolution - 1) | bottom > (int) (_y_resolution - 1))
        return;

    // Check to see if it overlaps a screen edge, and truncate if necessary

    if (left < 0)
        left = 0;

    if (bottom < 0)
        bottom = 0;

    if (right > (int) (_x_resolution - 1))
        right = (int) (_x_resolution - 1);
    if (top > (int) (_y_resolution - 1))
        top = (int) (_y_resolution -1);

    // Add it

    assert (_super_pixel_index < _max_super_pixels);

    SuperPixel *super_pixel = &_super_pixels[_super_pixel_index];

    super_pixel->x = (unsigned short) (left);
    super_pixel->y = (unsigned short) (bottom);
    super_pixel->width = (unsigned short) (right - left + 1);
    super_pixel->height = (unsigned short) (top - bottom + 1);

    ++_super_pixel_index;
}
INTEGRAT.CXX:
#include <assert.h>
#include <string.h>

#include "integrat.h"
#include "tga.h"
#include "yuv.h"

void Integrate (SuperPixel *super_pixels,
    unsigned super_pixel_count,
    void *source_image,
    unsigned image_width,
    Palette *palette,
    unsigned tga_image_type_code)
{
    for (unsigned super_pixel_index - 0; super_pixel_index < super_pixel_count; super_pixel_index++)
    {
        SuperPixel *super_pixel = &super_pixels[super_pixel_index];

        unsigned super_pixel_x = super_pixel->x;
        unsigned super_pixel_y = super_pixel->y;
        unsigned super_pixel_width = super_pixel->width;
        unsigned super_pixel_height = super_pixel->height;

        switch (tga_image_type_code)
        {
            case TGA_COLOR_MAPPED:
            {
                RGB rgb_temp;
                RGB rgb_avg;

                rgb_avg.r = 0;
                rgb_avg.g = 0;
                rgb_avg.b = 0;

                for (unsigned y_index = 0; y_index < super_pixel_height; y_index++)
                {
```

```
            unsigned y_offset = (y_index + super_pixel_y) * image_width + super_pixel_x;

            for (unsigned x_index = 0; x_index < super_pixel_width; x_index++)
            {
               rgb_temp = palette->Get (((char *) source_image)[y_offset + x_index]);

               rgb_avg.r += rgb_temp.r;
               rgb_avg.g += rgb_temp.g;
               rgb_avg.b += rgb_temp.b;
            }
         }

         rgb_avg.r /= (super_pixel_height * super_pixel_width);
         rgb_avg.g /= (super_pixel_height * super_pixel_width);
         rgb_avg.b /= (super_pixel_height * super_pixel_width);

         int color = YUVFindClosestRGB (rgb_avg, *palette);

         assert (color < 256 && color >= 0);

         super_pixel->color = color;
      }
      break;

      case TGA_TRUE_COLOR:
      {
         RGB rgb avg;
         rgb_avg.r = 0;
         rgb_avg.g = 0;
         rgb_avg.b = 0;

         unsigned long color;

         for (unsigned y_index = 0; y_index < super_pixel_height; y_index++)
         {
            unsigned y_offset = (y_index + super_pixel_y) * image_width + super_pixel_x;

            for (unsigned x_index = 0; x_index < super_pixel_width; x_index++)
            {
               color = *(((long *) source_image) + y_offset + x_index);

               rgb_avg.r += (color & 0x00FF0000) >> 16;
               rgb_avg.g += (color & 0x0000FF00) >> 8;
               rgb_avg.b += color & 0x000000FF;
            }
         }

         rgb_avg.r /= (super_pixel_height * super_pixel_width);
         rgb_avg.g /= (super_pixel_height * super_pixel_width);
         rgb_avg.b /= (super_pixel_height * super_pixel_width);

         color = rgb_avg.r << 16;
         color += rgb_avg.g << 8;
         color += rgb_avg.b;
```

```
                assert (color < 0x01000000);

                super_pixel->color = color;
            }
            break;

            case TGA_GREY_SCALE:
            {
                int color = 0;

                for (unsigned y_index = 0; y_index < super_pixel_height; y_index++)
                {
                    unsigned y_offset = (y_index + super_pixel_y) * image_width + super_pixel_x;

                    for (unsigned x_index = 0; x_index < super_pixel_width; x_index++)
                        color += ((char *) source_image)[y_offset + x_index];
                }

                color /= (super_pixel_height * super_pixel_width);

                assert (color < 256 && color >= 0);

                super_pixel->color = color;
            }
            break;

            default;
            throw "Invalid type code";
        }
    }
}
INTERPOL.CXX:
#include <string.h>

#include "interpol.h"
#include "tga.h"

void Interpolate (SuperPixel *super_pixels,
    unsigned super_pixel_count,
    char *dest_image,
    unsigned image_width,
    unsigned tga_image_type_code)
{
    for (unsigned super_pixel_index = 0; super_pixel_index < super_pixel_count; super_pixel_index++)
    {
        SuperPixel *super_pixel = &super_pixels[super_pixel_index];

        unsigned super_pixel_x = super_pixel->x;
        unsigned super_pixel_y = super_pixel->y;
        unsigned super_pixel_width = super_pixel->width;
        unsigned super_pixel_height = super_pixel->height;
        unsigned long super_pixel_color = super_pixel->color;
        switch (tga_image_type_code)
        {
            case TGA_COLOR_MAPPED:
            {
                for (unsigned y_index = 0; y_index < super_pixel_height; y_index++)
                {
                    unsigned offset = (y_index + super_pixel_y) * image_width + super_pixel_x;

                    memset ((void *) (&dest_image[offset]), super_pixel_color, super_pixel_width);

                }
            }
            break;

            case TGA_TRUE_COLOR:
            {
                for (unsigned y_index = 0; y_index < super_pixel_height; y_index++)
                {
                    unsigned offset = (y_index + super_pixel_y) * image_width + super_pixel_x;

                    for (unsigned x_index = 0; x_index < super_pixel_width; x_index++)
                        *(((long *) dest_image) + offset + x_index) = super_pixel_color;
                }
            }
            break;

            case TGA_GREY_SCALE:
            {
                for (unsigned y_index = 0; y_index < super_pixel_height; y_index++)
                {
                    unsigned x_offset = (y_index + super_pixel_y) * image_width + super_pixel_x;

                    memset ((void *) (&dest_image[x_offset]), super_pixel_color, super_pixel_width);
                }
            }
            break;

            default:
            throw "Invalid type code";
        }
    }
}
```

LEVEL.CXX:

```
#include <string.h>

#include "assert.h"
#include "level.h"

void Reduce2x2 (Level *a, Level *b)
{
    // Average blocks in a and store them in b
    unsigned width = a->width;
    unsigned height = a->height;
    char *image_a = a->image;
    char *image_b = b->image;

    for (unsigned y_index = 0; y_index < height - 1; y_index += 2)
    {
        for (unsigned x_index = 0; x_index < width - 1; x_index += 2)
        {
            assert (y_index * width + x_index < a->width * a->height);
            assert (y_index * width + x_index + 1 < a->width * a->height);
            assert ((y_index + 1) * width + x_index < a->width * a->height);
            assert ((y_index + 1) * width + x_index + 1 < a->width * a->height);

            int avg = image_a[y_index * width + x_index];

            avg += image_a[y_index * width + x_index + 1];
            avg += image_a[(y_index + 1) * width + x_index];
            ave += image_a[(y_index + 1) * width + x_index + 1];

            avg /= 4;

            assert (y_index / 2 * width / 2 + x_index / 2 < b->width * b->height);

            image_b[y_index / 2 * width / 2 + x_index / 2] = (char) (avg);
        }
    }
}

void Expand2x2 (Level *a, Level *b, Threshold *threshold)
{
    // Interpolate pixels in 'a' and store them in 'b'

    // 'b' is stored as a sequential block, with the top left
    // pixel starting in b's image buffer at index 0.

    unsigned x1;
    unsigned y1;
    unsigned x2;
    unsigned y2;

    if (threshold)
    {
        // Only interpolate in 'b' over the region from (x1, y1) to (x2, y2)

        x1 = threshold->x1;
        y1 = threshold->y1;
        x2 = threshold->x2;
        y2 = threshold->y2;
    }
    else
    {
        // Do the entire block
    x1 = 0;
    y1 = 0;
    x2 = b->width;
    y2 = b->height;
}

unsigned b_width = x2 - x1;
unsigned b_height = y2 - y1;

// First do all of the inside 2 x 2 blocks

for (unsigned y_index = 1; y_index < b_height - 1; y_index += 2)
{
    for (unsigned x_index = 1; x_index < b_width - 1; x_index += 2)
    {
        unsigned a_index = (y1 + y_index) / 2 * a->width + (x1 + x_index) / 2;

        assert (a_index < a->width * a->height);

        char *pl = &a->image[a_index];

        char v1 = (char) ((*pl) / 2);
        char v2 = (char) ((*(pl + 1)) / 2);
        char v3 = (char) ((*(p1 + a->width)) / 2);
        char v4 = (char) ((*(p1 + a->width + 1)) / 2);

        unsigned b_index = y_index * b_width + x_index;

        assert (b_index + b_width + 1 < b->width * b->height);

        char *p2 = &b->image[b_index];

        *p2 =           (char) (v1 + (v2  + v3) / 2);
        *(p2 + 1) =       (char) (v2 + (v1 + v4) / 2);
        *(p2 + b_width) =   (char) (v3 + (v1 + v4) / 2);
        *(p2 + b_width + 1) = (char) (v4 + (v2 + v3) / 2);
    }
}

// Now do the 1 pixel wide edge around the whole thing

for (unsigned x_index = 0; x_index < b_width; x_index++)
{
    // Top and bottom

    assert (x_index < b->width * b->height);
    assert ((b_height - 1) * b_width + x_index < b->width * b->height);

    b->image[x_index] = b->image[b_width + x_index];
    b->image[(b_height - 1) * b_width + x_index] =
        b->image[(b_height - 2) * b_width + x_index];
}

for (y_index = 0; y_index < b_height; y_index++)
```

```
    {
        // Left and right

        assert (y_index * b_width < b->width * b->height);
        assert (y_index * b_width + (b_width - 1) < b->width * b->height);

        b->image[y_index * b_width] = b->image[y_index * b_width + 1];
        b->image[y_index * b_width + (b_width - 1)] =
            b->image[y_index * b_width + (b_width - 2)];
    }
}

void Reduce3x3 (Level *a, Level *b)
{
    Reduce2x2 (a, b);
}

void Expand3x3 (Level *a, Level *b, Threshold *threshold)
{
    Expand2x2 (a, b, threshold);
}

void Subtract (Level *a, Level *b, Threshold *threshold)
{
    // b = a - b

    unsigned x1;
    unsigned y1;
    unsigned x2;
    unsigned y2;

    if (threshold)
    {
        x1 = threshold->x1;
        y1 = threshold->y1;
        x2 = threshold->x2;
        y2 = threshold->y2;
}
else
{
    x1 = 0;
    y1 = 0;
    x2 = a->width;
    y2 = a=>height;
}

// The difference is stored in 'b' sequentially

unsigned b_index = 0;

for (unsigned y = y1; y < y2; y++)
{
    for (unsigned x = x1; x < x2; x++)
    {
            assert (b_index < b->width * b->height);
            assert (y * a->width + x < a->width * a->height);

            b->image[b_index] =
                (char) (a->image[y * a->width + x] - b->image[b_index]);

            ++b_index;
        }
    }

    if (threshold)
    {
        unsigned b_width = x2 - x1;
        unsigned b_height = y2 - y1;

        for (unsigned x_index = 0; x_index < b_width; x_index++)
        {
            // Top and bottom

            assert (x_index < b->width * b->height);
            assert ((b_index - 1) * b_width + x_index < b->width * b->height);

            b->image[x_index] - 0;
            b->image[(b_height - 1) * b_width + x_index] = 0;
        }

        for (unsigned y_index = 0; y_index < b_height; y_index++)
        {
            // Left and right

            assert (y_index * b_width < b->width * b->height);
            assert (y_index * b_width + (b_width - 1) < b->width * b->height);

            b->image[y_index * b_width] = 0;
            b->image[y_index * b_width + (b_width - 1)] = 0;
        }
    }
}

void Add (Level *a, Level *b, Threshold *threshold)
{
    // b = a + b

    unsigned x1;
    unsigned y1;
    unsigned x2;
    unsigned y2;

    if (threshold)
    {
        x1 - threshold->x1;
        y1 - threshold->y1;
        x2 - threshold->x2;
        y2 - threshold->y2;
    }
    else
    {
        x1 = 0;
        y1 = 0;
        x2 = b->width;
        y2 = b->height;
    }

    // The contents of 'a' are stored sequentially

    unsigned a_index = 0;

    for (unsigned y = y1; y < y2; y++)
    {
        for (unsigned x = x1; x < x2; x++)
        {
            assert (a_index < a->width * a->height);
            assert (y * b->width + x < b->width * b->height);

            b->image[y * b->width + x] += a->image[a_index];

            ++a_index;
        }
    }
}
void Copy (Level *a, Level *b)
{
    b->width = a->width;
    b->height = a->height;

    memcpy (b->image, a->image, a->width * a->height);
}

MOUSE.CXX:
#include <dos.h>

#include "mouse.h"

int MouseInit ()
{
    REGS regs;

    // SW Mouse reset

    regs.w.ax = 0x21;
    int386 (0x33, &regs, &regs);

    // Mouse init

    regs.w.ax = 0x00;
    int386 (0x33, &regs, &regs);

    return (regs.w.ax == 0x21); // return 0 on success
}

unsigned short MouseButtonStatus ()
{
    REGS regs;

    regs.w.ax = 0x03;

    int386 (0x33, &regs, &regs);

    return regs.w.bx;
}

unsigned long MousePosition ()
{
    REGS regs;

    regs.w.ax = 0x03;

    int386 (0x33, &regs, &regs);

    // Return column in upper word and row in lower word

    return ((unsigned long) regs.w.cx) << 16 | regs.w.dx;
}

unsigned long MouseMotion ()
{
    REGS regs;

    regs.w.ax = 0x0B;

    int386 (0x33, &regs, &regs);

    // This returns the number of mickeys moved since the last call.

    // A mickey is the smallest unit a mouse can move.

    // The upper word contains movement in the x direction, and
    // the lower word contains movement in the y direction.

    // Note that the motion can be a negative value.

    return ((unsigned long) regs.w.cx) << 16 | regs.w.dx;
}
PALETTE.CXX:
#include <math.h>

#include "palette.h"

Palette::Palette (unsigned size, unsigned bits_per_primary)
```

```
{
   _size = size;
   _bits_per_primary = bits_per_primary;
   _rgb_table = new RGB [size];

   if (!_rgb_table)
      throw "Palette::Palette(): Memory allocation error";

   LoadGreyScales ();
}

Palette::~Palette ()
{
   delete _rgb_table;
}

void Palette::Set (unsigned index, unsigned grey_scale)
{
   if (index >= _size)
      throw "Palette::Set(): Color index out of range";

   _rgb_table[index].r =
   _rgb_table[index].g =
   _rgb_table[index].b = grey_scale;
}

void Palette::Set (unsigned index, unsigned r, unsigned g, unsigned b)
{
   if (index >= _size)
      throw "Palette::Set(): Color index out of range";

   _rgb_table[index].r = r;
   _rgb_table[index].g = g;
   _rgb_table[index].b = b;
}

void Palette::Set (unsigned index, RGB &rgb)
{
   if (index >= _size)
      throw "Palette::Set(): Color index out of range";

   _rgb_table[index].r = rgb.r;
   _rgb_table[index].g = rgb.g;
   _rgb_table[index].b = rgb.b;
}

RGB &Palette::Get (unsigned index)
{
   if (index >= _size)
      throw "Palette::Get(): Color index out of range";

   return _rgb_table[index];
}
```

```
void Palette::LoadGreyScales ()
{
    for (unsigned index = 0; index <_size; index++)
    {
        unsigned color = (unsigned long) index * (1 <<_bits_per_primary) / _size;

        _rgb_table[index].r =
        _rgb_table[index].g =
        _rgb_table[index].b = color;
    }
}
PROMPT.CXX:
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "prompt.h"

void GetString (char *string)
{
    // Read a string from stdin, but don't overwrite
    // argument if nothing is entered.

    char buffer[80];

    fgets (buffer, 80, stdin);

    if (strlen (buffer))
        buffer[strlen(buffer) - 1] = '\0'; // delete <CR>

    if (strlen (buffer) > 0)
        strcpy (string, buffer);
}

void GetFloat (float &result)
{
    // Read a float from stdin, but don't overwrite
    // argument if nothing is entered.

    char buffer[80];

    fgets (buffer, 80, stdin);

    float temp = atof (buffer);

    if (temp)
        result = temp;
}

void GetUnsigned (unsigned &result)
{
    // Read an unsigned from stdin, but don't overwrite
    // argument if nothing is entered.
    char buffer[80];

    fgets (buffer, 80, stdin);

    unsigned nTemp = atoi (buffer);

    if (nTemp)
        result = nTemp;
}
PTIMER.CXX:
#include "ptimer.h"

static unsigned long start_high, start_low;
static unsigned long stop_high, stop_low;

extern void asm_start_timer ();
extern void asm_stop_timer ();

// 0F3 1h is the Pentium ReaD Time Stamp Counter (RDTSC) Instruction

#pragma aux asm_start_timer = \
        "db 0Fh"        \
        "db 3 1h"       \
        "mov start_high,edx"  \
        "move start_low,eax"  \
        modify [edx eax];
#pragma aux asm_stop_timer = \
        "db 0Fh"        \
        "db 3 1h"       \
        "move stop_high,edx"  \
        "move stop_low,eax"   \
        modify [edx eax];

PentiumTimer::PentiumTimer ()
{
    started_flag = 0;
}

void PentiumTimer::Start ()
{
    started_flag = 1;

    asm_start_timer ();
}

void PentiumTimer::Stop ()
{
    started_flag = 0;

    asm_stop_timer ();
}

void PentiumTimer::Elapsed (unsigned long &high, unsigned long &low)
{
    if (started_flag)
        asm_stop_timer ();

    high = stop_high - start_high;

    if (stop_low > start_low)
        low = stop_low - start_low;
    else
    {
        // If the low part has looped, borrow from the high part

        low = 0xFFFFFFFF + (stop_low - start_low);
        high -= 1;
    }
}

unsigned long PentiumTimer::HighReg ()
{
    asm_stop_timer ();

    return stop_high;
}

unsigned long Pentium Timer::LowReg ()
{
    asm_stop_timer ();

    return stop_low;
}
PYRAMID.CXX:
#include <assert.h>
#include <math.h>
#include <string.h>

#include "pyramid.h"

Pyramid::Pyramid (unsigned base_width,
    unsigned base_height,
    unsigned levels,
    float contrast_threshold,
    float half_resolution,
    float dominant_freq0,
    float image_distance,
    float pixel_per_cm)
{
    if (!(base_width >> (levels - 1)))
        throw "Pyramid::Pyramid(): Too many levels specified for this width";

    if (!(base_height >> (levels - 1)))
        throw "Pyramid::Pyramid(): Too many levels specified for this height";

    _base_width = base_width;
    _base_height = base_height;
    if (levels < 2)
        throw "Pyramid::Pyramid(): You must specify more than 1 level";

    _levels = levels;
    _contrast_threshold = contrast_threshold;
    _half_resolution = half_resolution;
    _dominant_freq0 = dominant_freq0;
    _image_distance = image_distance;
    _pixels_per_cm = pixels_per_cm;

    _big_levels = 0;
    _small_levels = 0;

    // Level N does not get thresholded

    _thresholds = new Threshold [levels - 1];

    if (!_thresholds)
        throw "Pyramid::Pyramid(): memory allocation error";
}

Pyramid::~Pyramid ()
{
    delete [] _thresholds;
}

void Pyramid::ComputeThresholds (unsigned x, unsigned y)
{
    if (_big_levels == 0)
        throw "Pyramid::Threshold(): levels have not been allocated yet";

    // The last level does not get thresholded

    for (unsigned level = 0; level < _levels - 1; level++)
    {
        int max_ecc = MaxEcc (level) >> level;

        if (max_ecc < 2)
            throw "Pyramid::ComputeThresholds(): An image has been thresholded beyond the lower limit";

        int x1 = (x >> level) - max_ecc;
        int y1 = (y >> level) - max_ecc;
        int x2 = (x >> level) + max_ecc;
        int y2 = (y >> level) + max_ecc;

        // Align thresholds on even boundaries to ensure that levels
        // will expand properly.  Otherwise, a roundoff can cause two
        // levels to be off by one pixel.

        if (x1 & 1)
            ++x1;

        if (y1 & 1)
            ++y1;
        if (x2 & 1)
            --x2;

        if (y2 & 1)
            --y2;

        // Clip the thresholded region on the image boundaries

        if (x1 < 0)
            _thresholds[level].x1 = 0;
        else
            _thresholds[level].x1 = x1;

        if (y1 < 0)
            _thresholds[level].y1 = 0;
        else
            _thresholds[level].y1 = y1;

        if (x2 > _big_levels[level].width)
            _thresholds[level].x2 = _big_levels[level].width;
        else
            _thresholds[level].x2 = x2;

        if (y2 > _big_levels[level].height)
            _thresholds[level].y2 = _big_levels[level].height;
        else
            _thresholds[level].y2 = y2;
    }
}

unsigned Pyramid::MaxEcc (unsigned level)
{
    const float NEG_ALPHA = -0.005;

    float max_ecc = (_half_resolution / (_dominant_freq0 / (1 << level))) *
        sqrt (log (_contrast_threshold) / NEG_ALPHA) - _half_resolution;

    const float PI = 2 * asin (1.0);

    max_ecc = max_ecc * tan (2 * PI / 360) * _image_distance * _pixels_per_cm;

    return max_ecc;
}

EncodePyramid::EncodePyramid (unsigned base_width,
    unsigned base_height,
    unsigned levels,
    float contrast_threshold,
    float half_resolution,
    float dominant_freq0,
    float image_distance,
    float pixels_per_cm) :
Pyramid (base_width,
     base_height
     levels,
     contrast_threshold,
     half_resolution,
     dominant_freq0,
     image_distance,
     pixels_per_cm)
{
}

EncodePyramid::~EncodePyramid ()
{
    // Deallocate all of the levels, but only if AllocateLevels was called

    if (!_big_levels)
        return;

    // Don't deallocate the image in level 0 because this
    // was allocated externally.

    for (unsigned level = 1; level < _levels; level++)
    {
        if (_big_levels[level].image)
            delete [] _big_levels[level].image;

        if (_small_levels[level].image)
            delete [] _small_levels[level].image;
    }

    delete [] _big_levels;
    delete [] _small_levels;
}

void EncodePyramid::AllocateLevels (Level *base_level)
{
    // Allocate the level structures

    _big_levels = new Level [_levels];
    _small_levels = new Level [_levels];

    if (!_big_levels || !_small_levels)
        throw "EncodePyramid::AllocateLevels(): memory allocation error";

    // Allocate all the image buffers within the levels and set their
    // correct sizes

    // When encoding, the lowest big level's image is not allocated,
    // instead, the image is external.

    _big_levels[0].image = base_level->image;
    _big_levels[0].width = base_level->width;
    _big_levels[0].height = base_level->height;

    for (unsigned level = 1; level < _levels; level++)
    {
        _big_levels[level].width = _base_width >> level;
        _big_levels[level].height = _base_height >> level;

        _big_levels[level].image =
            new char [_big_levels[level].width * _big_levels[level].height];

        if (!_big_levels[level].image)
            throw "EncodePyramid::AllocateLevels(): memory allocation error";
    }

    // When encoding, the highest small level is not used

    _small_levels[_levels - 1].width = 0;
    _small_levels[_levels - 1].height = 0;
    _small_levels[_levels - 1].image = 0;

    for (level = 0; level < _levels - 1; level++)
    {
        _small_levels[level].width = _base_width >> level;
        _small_levels[level].height = _base_height >> level;

        // When encoding, the small image buffers can't be
        // bigger than the largest thresholded region.

        if (((MaxEcc (level) * 2) >> level) < _small_levels[level].width)
            _small_levels[level].width = (MaxEcc (level) * 2) >> level;

        if (((MaxEcc (level) * 2) >> level) < _small_levels[level].height)
            _small_levels[level].height = (MaxEcc (level) * 2) >> level;

        _small_levels[level].image =
            new char [_small_levels[level].width * _small_levels[level].height];

        if (!_small_levels[level].image)
            throw "EncodePyramid::AllocateLevels(): memory allocation error";
    }
}

void EncodePyramid::Reduce (unsigned level)
{
    // Reduce level to level + 1

    if (level >= _levels - 1)
        throw "EncodePyramid::Reduce(): level is out of range";

    ::Reduce2x2 (&_big_levels[level], &_big_levels[level + 1]);
}

void EncodePyramid::Expand (unsigned level)
{
    // Expand level to level - 1

    if (level >= _levels)
```

```
        throw "EncodePyramid::Expand(): level is out of range";

    if (level < 1)
        throw "EncodePyramid::Expand(): level is out of range";

    ::Expand2x2 (&_big_levels[level], &_small_levels[level - 1], &_thresholds[level -1]);
}

void EncodePyramid::Subtract (unsigned level)

{
    // Subtract the big level from the small level and store the
    // result in the small level

    if (level >= _levels - 1)
        throw "EncodePyramid::Subtract(): level is out of range";

    ::Subtract (&_big_levels[level], &_small_levels[level], &_thresholds[level]);
}

Level *EncodePyramid::GetLevel (unsigned level)
{
    // Return small level

    if (level >= _levels - 1)
        throw "EncodePyramid::Level(): level is out of range";

    return &_small_levels[level];
}

Level *EncodePyramid::GetLevelN ()
{
    // Return highest big level

    Level *level = &_big_levels[_levels - 1];

    return level;
}

DecodePyramid::DecodePyramid (unsigned base_width,
    unsigned base_height,
    unsigned levels,
    float contrast_threshold,
    float half_resolution,
    float dominant_freq0,
    float image_distance,
    float pixels_per_cm) :
Pyramid (base_width,
    base_height,
    levels,
    contrast_threshold,
    half_resolution,
    dominant_freq0,
    image_distance,
    pixels_per_cm)
```

```
{
}

DecodePyramid::~DecodePyramid ()
{
    // Deallocate all of the levels, but only if AllocateLevels was called

    if (!_big_levels)
        return;

    // No small levels are used when decoding

    for (unsigned level = 0; level < _levels; level++)
        delete [] _big_levels[level].image;

    delete [] _big_levels;
}

void Decode Pyramid::AllocateLevels ()
{
    // Allocate the level structures

    _big_levels = new Level [_levels];

    if (!_big_levels)
        throw "EncodePyramid::AllocateLevels(): memory allocation error";

    // When decoding, no small levels are used

    for (unsigned level = 0; level < _levels; level++)
    {
        _big_levels[level].width = _base_width >> level;
        _big_levels[level].height = _base_height >> level;

        _big_levels[level].image =
            new char [_big_levels[level].width * _big_levels[level].height];

        if (!_big_levels[level].image)
            throw "DecodePyramid::AllocateLevels(): memory allocation error";
    }
}

void DecodePyramid::Expand (unsigned level)
{
    // Expand level to level - 1

    if (level >= _levels)
        throw "DecodePyramid::Expand(): level is out of range";

    if (level == 0)
        throw "DecodePyramid::Expand(): level is out of range";

    ::Expand2x2 (&_big_levels[level], &_big_levels[level - 1]);
}
```

```
void DecodePyramid::Add (Level *differenced_level, unsigned level)
{
    // Add at the differenced level to the pyramid at the
    // specified level and store it in the big level

    if (level >= _levels)
        throw "DecodePyramid::Add(): level is out of range";

    ::Add (differenced_level, &_big_levels[level], &_thresholds[level]);
}

Level *Decode Pyramid::GetLevel (unsigned level)
{
    // Return big level

    if (level >= _levels - 1)
        throw "DecodePyramid::GetLevel(): level is out of range";

    return &_big_levels[level];
}

void DecodePyramid::SetLevelN (Level *level)
{
    // Set the topmost level of the pyramid

    if (level->width != _big_levels[_levels - 1].width)
        throw "DecodePyramid::SetLevelN(): width is incorrect";

    if (level->height != _big_levels[_levels - 1].height)
        throw "DecodePyramid::SetLevelN(): height is incorrect";

    memcpy (_big_levels[_levels - 1].image, level->image,
        level->width * level->height);
}
TGA.CXX:
#include <stdio.h>
#include <string.h>

#include "tga.h"

TGA::TGA ()
{
    _image = 0;
    _width = 0;
    _height = 0;
    memset (&_tga_header, 0, sizeof (_tga_header));
    _palette = 0;
}

TGA::~TGA ()
{
    if (_image)
        delete [] _image;
    if (_palette)
        delete _palette;
}

int TGA::Load (char *filename)
{
    FILE *tga_file = fopen (filename, "rb");

    if (!tga_file)
        return -1;

    _tga_header.id_field_length = (unsigned char) (fgetc (tga_file));
    _tga_header.color_map_type = (unsigned char) (fgetc (tga_file));
    _tga_header.image_type_code = (unsigned char) (fgetc (tga_file));
    _tga_header.color_map_origin = (unsigned short) ((fgetc (tga_file) & 0xFF) | (fgetc (tga_file) <<8));
    _tga_header.color_map_length = (unsigned short) ((fgetc (tga_file) & 0xFF) | (fgetc (tga_file) <<8));
    _tga_header.color_map_entry_size = (unsigned char) (fgetc (tga_file));
    _tga_header.x_origin = (unsigned short) ((fgetc (tga_file) & 0xFF) | (fgetc (tga_file) <<8));
    _tga_header.y_origin = (unsigned short) ((fgetc (tga_file) & 0xFF) | (fgetc (tga_file) <<8));
    _tga_header.width = (unsigned short) ((fgetc (tga_file) & 0xFF) | (fgetc (tga_file) <<8));
    _tga_header.height = (unsigned short) ((fgetc (tga_file) & 0xFF) | (fgetc (tga_file) <<8));
    _tga_header.bits_per_pixel = (unsigned char) (fgetc (tga_file));
    _tga_header.image_descriptor = (unsigned char) (fgetc (tga_file));

    for (unsigned index = 0; index <_tga_header.id_field_length; index++)
        fgetc (tga_file);

    switch (_tga_header.image_type_code)
    {
        case 0:
        throw "TGA::Load(): The image contains no data";

        case 1: // Color-mapped RGB
        case 2: // Indexed RGB
        case 3: // Greyscale
        break;

        case 9:
        case 10:
        case 11:
        throw "TGA::Load(): Compressed TGA files are not supported";

        default:
        throw "TGA::Load(): Unknown TGA file format is not supported";
    }

    if (_tga_header.color_map_type)
    {
        if (_tga_header.color_map_entry_size != 24)
            throw "TGA::Load(): For indexed RGB images, only 24 bit palettes are supported";

        if (_palette)
            delete _palette;
        _palette = new Palette (_tga_header.color_map_length);

        if (!_palette)
            throw "TGA::Load(): Memory allocation error";

        for (index = 0; index < _tga_header.color_map_length; index++)
        {
            unsigned nB = fgetc (tga_file);
            unsigned nG = fgetc (tga_file);
            unsigned nR = fgetc (tga_file);

            _palette->Set (index, nR, nG, nB);
        }
    }
    else
    {
        if (_palette)
            delete _palette;

        _palette = 0;
    }

    // Check to make sure the header was read in OK

    if (feof (tga_file) | ferror (tga_file))
    {
        fclose (tga_file);

        throw "TGA:Load(): Invalid TGA file";
    }

    if (_tga_header.image_descriptor & INTERLEAVE_MASK)
    {
        fclose (tga_file);

        throw "TGA::Load(): Interleaved images are not supported";
    }

    if (_image)
        delete [] _image;

    _image = 0;

    if (_tga_header.bits_per_pixel == 8)
        _image = new char [_tga_header.width * _tga_header.height];
    else if (_tga_header.bits_per_pixel == 24)
    {
        // If it's a 24 bit per pixel image, let's convert it to 32bpp
        // in memory so that we can do 32 bit image processing

        _tga_header.bits_per_pixel = 32;

        _image = new char [_tga_header.width * _tga_header.height * 4];
```

```
 }
 else
 {
     fclose (tga_file);

     throw "TGA::Load(): Only 8 or 24 bits per pixel images are supported";
 }

 if (!_image)
 {
     fclose (tga_file);

     throw "Memory allocation error";
 }

 if (_tga_header.image_descriptor & SCREEN_ORIGIN_MASK)
 {
     // Image is stored from top left to bottom right

     if (_tga_header.bits_per_pixel == 8)
     {
         for (index = 0; index < _tga_header.width * _tga_header.height; index++)
             _image[index] = (char) (fgetc (tga_file));
     }
     else
     {
         // 32 bpp

         for (index = 0; index < _tga_header.width * _tga_header.height * 4; index += 4)
         {
             unsigned long dwPixel;

             dwPixel = fgetc (tga_file);     // R
             dwPixel += fgetc (tga_file) << 8; // G
             dwPixel += fgetc (tga_file) << 16; // B

             (* (unsigned long *) (&_image[index])) = dwPixel;
         }
     }
 }
 else
 {
     // Image is stored from bottom left to top right

     if (_tga_header.bits_per_pixel == 8)
     {
         for (unsigned nHeight = _tga_header.height; nHeight > 0 ; nHeight--)
         {
             for (unsigned nWidth = 0; nWidth < __tga_header.width; nWidth++)
                 _image[(nHeight - 1) * _tga_header.width + nWidth] = (char) (fgetc (tga_file));
         }
     }
     else
     {
            // 32 bpp

            for (unsigned nHeight = _tga_header.height; nHeight > 0 ; nHeight--)
            {
                for (unsigned index = 0; index < _tga_header.width; index++)
                {
                    unsigned long dwPixel;

                    dwPixel = fgetc (tga_file);      // R
                    dwPixel += fgetc (tga_file) << 8; // G
                    dwPixel += fgetc (tga_file) << 16; // B

                    (* (unsigned long *) (&_image[(nHeight - 1) * _tga_header.width * 4 + index * 4])) = dwPixel;
                }
            }
        }
    }

    // Check to make sure the image was read in OK

    if (feof (tga_file) | ferror (tga_file))
    {
        fclose (tga_file);

        throw "Invalid TGA file";
    }
    fclose (tga_file);

    return 0; // return 0 on success
}
VESA.CXX:
#include <string.h>

#include "vesa.h"

static SV_devCtx *device_context = 0;
static int vesa_init_flag = 0;

int VESAInit (unsigned x, unsigned y, unsigned depth_bits, int bLinearBuffer)
{
    device_context = SV_init (true); // true = use VBE/AF if available

    if (!device_context | device_context->VBEVersion < 0x200)
        return -1;

    unsigned short *mode;

    for (mode = device_context->modeList; *mode != 0xFFFF; mode++)
    {
        SV_modeInfo ModeInfo;

        if (SV_getModeInfo (*mode, &ModeInfo) &&
            (!bLinearBuffer | (ModeInfo.Attributes & svHaveLinearBuffer)) &&
            ModeInfo.XResolution == x &&
            ModeInfo.YResolution == y &&
            ModeInfo.BitsPerPixel == depth_bits)
            break;
    }
    if (*mode == 0xFFFF)
        return -1;

    if (!SV_setMode (*mode | (unsigned short) (bLinearBuffer ? svLinearBuffer : 0),
        true, // 8BitDAC
        true, // virtual linear framebuffer
        1)) // # of multibuffer buffers
        return -1;

    vesa_init_flag = 1;

    return 0;
}

void VESAClose ()
{
    if (vesa_init_flag)
        SV_restoreMode ();

    vesa_init_flag = 0;
    device_context = 0;
}

const VESADeviceContext *VESAGetDeviceContext ()
{
    return device_context;
}

void VESALoad (Palette &palette)
{
    SV_palette *sv_palette = new SV_palette [palette.Size ()];

    if (!sv_palette)
        throw "VESALoadPalette(): Memory allocation error";

    // A VESA palette entry is always 8 bits per primary

    if (palette.BitsPerPrimary () != 8)
        throw "VESALoadPalette(): VESA palettes must have 8 bits per primary";


    // Loading palette entries one at a time doesn't work with
    // the dos32\wc\svga library dated 05/05/96. Instead, you
    // must load them all at once.

    for (unsigned index = 0; index < palette.Size (); index++)
    {
        RGB rgb;
        try
        {
            rgb = palette.Get (index);
        }
        catch (...)
        {
            delete [] sv_palette;

            throw;
        }

        sv_palette[index].red = (char) rgb.r;
        sv_palette[index].green = (char) rgb.g;
        sv_palette[index].blue = (char) rgb.b;
        sv_palette[index].alpha = 0;
    }

    SV_setPalette (0, palette.Size (), sv_palette, -1);

    delete [] sv_palette;
}

void VESALine (unsigned x1, unsigned y1, unsigned x2, unsigned y2, unsigned long color)
{
    SV_line (x1, y1, x2, y2, color);
}

void VESABeginLine ()
{
    SV_beginLine ();
    SV_beginDirectAccess ();
}

void VESAEndLine ()
{
    SV_endDirectAccess ();
    SV_endLine ();
}

void VESALineFast (unsigned x1, unsigned y1, unsigned x2, unsigned y2, unsigned long color)
{
    SV_lineFast (x1, y1, x2, y2, color);
}

void VESAPoint (unsigned x, unsigned y, unsigned long color)
{
    SV_putPixel (x, y, color);
}

void VESABeginPoint ()
{
    SV_beginDirectAccess ();
}
```

```
void VESAEndPoint ()
{
   SV_endDirectAccess ();
}

void VESAPointFast (unsigned x, unsigned y, unsigned long color)
{
   SV_putPixelFast (x, y, color);
}

void VESAText (char *szText, unsigned x, unsigned y, unsigned long color)
{
   SV_writeText (x, y, szText, color);
}

void VESAScreenBlt (char *bitmap, unsigned long length)
{
   if (!device_context)
      throw "VESAScreenBlt: VESA not initialized";

   if (!device.context->linearAddr)
      throw "VESAScreenBlt(): Not available in banked address mode";

   SV_beginDirectAccess ();

   memcpy ((char *) device_context->videoMem, bitmap, length);

   SV_endDirectAccess ();
}

void VESABitBlt (void *bitmap, unsigned width, unsigned height,
   unsigned bytes_per_pixel, unsigned x_offset, unsigned y_offset)
{
   if (!device_context)
      throw "VESABitBlt: VESA not initialized";

   if (!device_context->linearAddr)
      throw "VESABitBlt(): Not available in banked address mode";

   SV_beginDirectAccess ();

   switch (bytes_per_pixel)
   {
      case 1:
      {
         for (unsigned y = 0; y < height; y++)
            memcpy ((char *) device_context->videoMem +
               y_offset * (device_context->maxx + 1) +
               y * (device_context->maxx + 1) +
               x_offset,
               (char *) bitmap + y * width,
               width);
         break;
```

```
            }

            case 4:
            {
                for (unsigned y = 0; y < height; y++)
                    memcpy ((long *) device_context->videoMem +
                        y_offset * (device_context->maxx + 1) +
                        y * (device_context->maxx + 1) +
                        x_offset,
                        (long *) bitmap + y * width,
                        width * 4);
                break;
            }

            default:
            throw "VESABitBlt(): Invalid number of bytes per pixel";
        }

        SV_endDirectAccess ();
    }

    void VESAClear (unsigned long color)
    {
        SV_clear (color);
    }
YUV.CXX:
#include <assert.h>
#include <stdlib.h>

#include "yuv.h"

const unsigned YUV_MAP_LENGTH = 1 << (YWIDTH + UWIDTH + VWIDTH);
static unsigned yuv_map[YUV_MAP_LENGTH];

void YUVMapInit (Palette &palette)
{
    for (unsigned index = 0; index < YUV_MAP_LENGTH; index++)
        yuv_map [index] = ~0;

    for (index = 0; index < 256; index++)
    {
        RGB rgb_temp = palette.Get (index);

        YUV yuv_temp;

        RGBtoYUV (rgb_temp, yuv_temp);

        unsigned long packed_yuv = PackYUV (yuv_temp);

        assert (packed_yuv < YUV_MAP_LENGTH);

        yuv_map [packed_yuv] = index;
    }
}
```

```
unsigned YUVFindClosestRGB (RGB &rgb, Palette &palette)
{
  // RGB is any 24 bit RGB value

  // The return value is an index into Palette

  YUV yuv1;

  // Convert the RGB to 24 bit YUV

  RGBtoYUV (rgb, yuv1);

  // Scale down the 24 bit YUV to an index

  unsigned long yuv_index = PackYUV (yuv1);

  assert (yuv_index < YUV_MAP_LENGTH);

  // If the RGB index for this YUV has not been computed yet,
  // then compute it now

  if (yuv_map[yuv_index] == ~0)
  {
    unsigned closest_rgb_index = 0;
    unsigned closest_yuv_difference = ~0;

    assert (palette.Size ());

    for (unsigned index = 0; index < palette.Size (); index++)
    {
      YUV yuv2;

      RGBtoYUV (palette.Get (index), yuv2);

      // Compare the difference between our original 24 bit
      // YUV to the 24 bit YUV that corresponds with this
      // RGB palette entry.

      unsigned difference = YUVDifference (yuv1, yuv2);

      if (difference < closest_yuv_difference)
      {
        closest_yuv_difference = difference;
        closest_rgb_index = index;
      }
    }
    yuv_map [yuv_index] = closest_rgb_index;
  }

  return yuv_map[yuv_index];
}
```

```
void RGBtoYUV (RGB &rgb, YUV &yuv)
{
    yuv.y = rgb.r *  0.299 + rgb.g * 0.587 + rgb.b * 0.114;
    yuv.u = rgb.r *  -0.169 + rgb.g * -0.331 + rgb.b * 0.5  + 128;
    yuv.v = rgb.r *  0.5 + rgb.g * -0.419 + rgb.b * -0.081 + 128;
}

void YUVtoRGB (YUV &yuv, RGB &rgb)
{
    int y = yuv.y;
    int u = yuv.u - 128;
    int v = yuv.v - 128;

    rgb.r = y * 1.0 + u * 0.0  + v * 1.402;
    rgb.g = y * 1.0 + u * -0.344 + v * -0.714;
    rgb.b = y * 1.0 + u * 1.772 + v * 0.0 ;
}

void ClipRGB (RGB &rgb, unsigned max)
{
    if ((int) rgb.r < 0)
        rgb.r = 0;

    if (rgb.r > max)
        rgb.r = max;

    if ((int) rgb.g < 0)
        rgb.g = 0;

    if (rgb.g > max)
        rgb.g = max;

    if ((int) rgb.b < 0)
        rgb.b = 0;

    if (rgb.b > max)
        rgb.b = max;
}

unsigned long PackYUV (YUV &yuv)
{
    return (unsigned long)
        ((((yuv.y >> YSHIFT1) << YSHIFT2) & YMASK) +
        (((yuv.u >> USHIFT1) << USHIFT2) & UMASK) +
        (((yuv.v >> VSHIFT1) << VSHIFT2) & VMASK));
}

void UnPackYUV (unsigned long PackedYUV, YUV &yuv)

{
    yuv.y = ((PackedYUV & YMASK) >> YSHIFT2) << YSHIFT1;
    yuv.u = ((PackedYUV & UMASK) >> USHIFT2) << USHIFT1;
    yuv.v = ((PackedYUV & VMASK) >> VSHIFT2) << VSHIFT1;
}
```

```
unsigned long YUVDifference (YUV &yuv1, YUV yuv2)
{
    unsigned difference;

    difference = abs (yuv1.y - yuv2.y);
    difference += abs (yuv1.u - yuv2.u);
    difference += abs (yuv1.v - yuv2.v);

    return difference;
}

ASSERT.H:
#ifndef ASSERT_H
#define ASSERT_H

#undef assert

#ifdef NDEBUG
#define assert(__ignore) ((void)0)
#else
#define assert(expr) ((expr)?(void)0:assert2(0,#expr,__FILE__,__LINE__))
#endif

extern void assert2 (int value, char *expr, char *file, unsigned lineno);

#endif

CURSOR.H:
#ifndef CURSOR_H
#define CURSOR_H

class Cursor
{
   public:
      int X () { return _x; }
      int Y () { return _y; }
      void SetMaxX (int max_x) { _max_x = max_x; _x = _max_x / 2; }
      void SetMaxY (int max_y) { _max_y = max_y; _y = _max_y / 2; }
      void SetWidth (unsigned width) { _width = width; }
      void SetColor (unsigned long color1, unsigned long color2)
      { _color1 = color1; _color2 = color2; }
      void SetOffsetX (int offset_x) { _offset_x = offset_x; }
      void SetOffsetY (int offset_y) { _offset_y = offset_y; }
      void UpdateX (int x);
      void UpdateY (int y);
      void Draw (void (*LineDraw) (unsigned x1, unsigned y1, unsigned x2, unsigned y2, unsigned long color));

   private:
      int_x;
      int_y;
      int_max_x;
      int_max_y;
      int_offset_x;
      int_offset_y;
```

```
        unsigned_width;
        unsigned long_color1;
        unsigned long_color2;
};

#endif
FOVEATOR.H:
#ifndef FOVEATOR_H
#define FOVEATOR_H

#include "superpix.h"

// This class will create an array of superpixels

class Foveator
{
    public:
        Foveator ();
        ~Foveator ();
        void SetScreenResolution (unsigned x_resolution, unsigned y_resolution);
        void SetHalfResolution (float half_resolution)
        { _half_resolution = half_resolution; }
        void SetDistance (unsigned distance)
        { _distance = distance; }
        void SetPixelsPerCM (unsigned pixels_per_cm)
        { _pixels_per_cm = pixels_per_cm; }
        void SetW0 (unsigned w0)
        { _w0 = w0; }

        SuperPixel *Foveate (unsigned center_x,
            unsigned center_y,
            unsigned &super_pixel_count);

        private:
            void AddSuperPixels (unsigned center_x, unsigned center_y,
                        unsigned x, unsigned y,
                        unsigned width, unsigned height);
            void AddSuperPixel (int x, int y, unsigned width, unsigned height);
            float _half_resolution;
            unsigned _distance;
            unsigned _pixels_per_cm;
            unsigned _w0;
            unsigned _x_resolution;
            unsigned _y_resolution;
            SuperPixel *_super_pixels;
            unsigned _super_pixel_index;
            unsigned _max_super_pixels;
};

#endif
INTEGRAT.H:
#ifndef INTEGRATE_H
#define INTEGRATE_H
```

```
#include "palette.h"
#include "superpix.h"

extern void Integrate (SuperPixel *super_pixels,
    unsigned super_pixel_count,
    void *source_image,
    unsigned image_width,
    Palette *palette,
    unsigned tga_image_type_code);

#endif
INTERPOL.H:
#ifndef INTERPOLATE_H
#define INTERPOLATE_H

#include "superpix.h"

extern void Interpolate (SuperPixel *super_pixels,
    unsigned super_pixel_count,
    char *dest_image,
    unsigned image_width,
    unsigned tga_image_type_code);

#endif

LEVEL.H:
#ifndef LEVEL_H
#define LEVEL_H

// A Level struct represents a level in a Laplacian Pyramid

struct Level
{
    unsigned width;
    unsigned height;
    char *image;
};

// A Threshold struct confines the contrast threshold region
// for a level. The region is not inclusive of x2,y2.

struct Threshold
{
    unsigned x1;
    unsigned y1;
    unsigned x2;
    unsigned y2;
};

// Reduce 'a' to 'b'

extern void Reduce2x2 (Level *a, Level *b);

// Expand 'a' to 'b'
```

```
extern void Expand2x2 (Level *a, Level *b, Threshold *threshold = 0);

// Reduce 'a' to 'b'

extern void Reduce3x3 (Level *a, Level *b);

// Expand 'a' to 'b'

extern void Expand3x3 (Level *a, Level *b, Threshold *threshold = 0);

// b = a - b

extern void Subtract (Level *a, Level *b, Threshold *threshold = 0);

// b = a + b

extern void Add (Level *a, Level *b, Threshold *threshold = 0);

// b = a

extern void Copy (Level *a, Level *b);

#endif
MOUSE.H:
#ifndef MOUSE_H
#define MOUSE_H

#define MOUSE_LEFT_BUTTON_MASK    0x01
#define MOUSE_RIGHT_BUTTON_MASK   0x02
#define MOUSE_MIDDLE_BUTTON_MASK  0x04

extern int MouseInit ();
extern unsigned short MouseButtonStatus ();
extern unsigned long MousePosition (); // 31-16=column, 15-0=row
extern unsigned long MouseMotion (); // 31-16=xdir, 15-0=ydir

#endif
PALETTE:H:
#ifndef PALETTE_H
#define PALETTE_H

#include "rgb.h"

class Palette
{
    public:
        Palette (unsigned size = 256, unsigned bits_per_primary = 8);
        ~Palette ();
        void Set (unsigned index, unsigned grey_scale_value);
        void Set (unsigned index, unsigned r, unsigned g, unsigned b);
        void Set (unsigned index, RGB &rgb);
        RGB &Get (unsigned index);
        unsigned Size () { return_size; }
        unsigned BitsPerPrimary () { return _bits_per_primary; }

    protected:
        RGB *_rgb_table;

    private:
        void LoadGreyScales ();
        unsigned _size;
        unsigned _bits_per_primary;

};

#endif
PROMPT.H:
#ifndef PROMPT_H
#define PROMPT_H

void GetString (char *string);
void GetFloat (float &result);
void GetUnsigned (unsigned &result);

#endif
PTIMER.H:
#ifndef PTIMER_H
#define PTIMER_H

// For some reason, the Watcom compiler generates a W627 warning
// whenever this header is included.  The following pragma will
// turn off this warning.

#pragma warning 627 9

class Pentium Timer
{
    public:
        Pentium Timer ();
        void Start ();
        void Stop ();
        void Elapsed (unsigned long &high, unsigned long &low);
        unsigned long HighReg ();
        unsigned long LowReg ();

    private:
        int started_flag;
};

#endif
PYRAMID.H:
#ifndef PYRAMID_H
#define PYRAMID_H

#include "level.h"

class Pyramid
```

```
{
    public:
        Pyramid (unsigned base_width,
            unsigned base_height,
            unsigned levels,
            float contrast_threshold,
            float half_resolution,
            float dominant_freq0,
            float image_distance,
            float pixels_per_cm);
        ~Pyramid ();
        void ComputeThresholds (unsigned x, unsigned y);

    protected:
        unsigned MaxEcc (unsigned level);

        unsigned _levels;
        unsigned _base_width;
        unsigned _base_height;
        Level *_big_levels;
        Level *_small_levels;
        Threshold *_thresholds;
        float _contrast_threshold;
        float _half_resolution;
        float _dominant_freq0;
        float _image_distance;
        float _pixels_per_cm;
};

class EncodePyramid : public Pyramid
{
    public:
        EncodePyramid (unsigned base_width,
            unsigned base_height,
            unsigned levels,
            float contrast_threshold,
            float half_resolution,
            float dominant_freq0,
            float image_distance,
            float pixels_per_cm);
        ~EncodePyramid ();
        void AllocateLevels (Level *base_level);
        void Reduce (unsigned level);
        void Expand (unsigned level);
        void Subtract (unsigned level);
        Level *GetLevel (unsigned level);
        Level *GetLevelN ();
};

class DecodePyramid : public Pyramid
{
    public:
        DecodePyramid (unsigned base_width,
            unsigned base_height,
            unsigned levels,
            float contrast_threshold,
            float half_resolution,
            float dominant_freq0,
            float image_distance,
            float pixels_per_cm);
        ~DecodePyramid ();
        void AllocateLevels ();
        void Expand (unsigned level);
        void Add (Level *differenced_level, unsigned level);
        Level *GetLevel (unsigned level);
        void SetLevelN (Level *level);
};

#endif
RGB.H:
#ifndef RGB_H
#define RGB_H

struct RGB
{
    unsigned r;
    unsigned g;
    unsigned b;
};

#endif

SUPERPIX.H:
#ifndef SUPERPIX_H
#define SUPERPIX_H

struct SuperPixel
{
    // If the origin is at the bottom left of your view, then
    // X and Y are the bottom left of the super pixel.

    unsigned x;
    unsigned y;
    unsigned width;
    unsigned height;
    unsigned long color;
};

#endif
TGA.H:
#ifndef TGA_H
#define TGA_H

#include "palette.h"

// Image descriptor byte masks

#define SCREEN_ORIGIN_MASK 0x20
```

```
#define INTERLEAVE_MASK    0xC0

/*
** FILE STRUCTURE FOR THE ORIGINAL TRUEVISION TGA FILE
**     FIELD 1 :   NUMBER OF CHARACTERS IN ID FIELD (1 BYTES)
**     FIELD 2 :   COLOR MAP TYPE (1 BYTES)
**     FIELD 3 :   IMAGE TYPE CODE (1 BYTES)
**             = 0  NO IMAGE DATA INCLUDED
**             = 1  UNCOMPRESSED, COLOR-MAPPED IMAGE
**             = 2  UNCOMPRESSED, TRUE-COLOR IMAGE
**             = 3  UNCOMPRESSED, BLACK AND WHITE IMAGE
**             = 9  RUN-LENGTH ENCODED COLOR-MAPPED IMAGE
**             = 10 RUN-LENGTH ENCODED TRUE-COLOR IMAGE
**             = 11 RUN-LENGTH ENCODED BLACK AND WHITE IMAGE
**     FIELD 4 :   COLOR MAP SPECIFICATION  (5 BYTES)
**           4.1 :  COLOR MAP ORIGIN (2 BYTES)
**           4.2 :  COLOR MAP LENGTH (2 BYTES)
**           4.3 :  COLOR MAP ENTRY SIZE (2 BYTES)
**     FIELD 5 :   IMAGE SPECIFICATION (10 BYTES)
**           5.1 :  X-ORIGIN OF IMAGE (2 BYTES)
**           5.2 :  Y-ORIGIN OF IMAGE (2 BYTES)
**           5.3 :  WIDTH OF IMAGE (2 BYTES)
**           5.4 :  HEIGHT OF IMAGE (2 BYTES)
**           5.5 :  IMAGE PIXEL SIZE (1 BYTE)
**           5.6 :  IMAGE DESCRIPTOR BYTE (1 BYTE)
**               BITS 0 -3 :     ATTRIBUTE BITS PER PIXEL
**               BIT 4 :      LEFT TO RIGHT PIXEL ORDERING
**               BIT 5 :      TOP TO BOTTOM PIXEL ORDERING
**               BIT 6 :      0
**               BIT 7 :      0
**     FIELD 6 :   IMAGE ID FIELD (LENGTH SPECIFIED BY FIELD 1)
**     FIELD 7 :   COLOR MAP DATA (BIT WIDTH SPECIFIED BY FIELD 4.3 AND
**           NUMBER OF COLOR MAP ENTRIES SPECIFIED IN FIELD 4.2)
**     FIELD 8 :   IMAGE DATA FIELD (WIDTH AND HEIGHT SPECIFIED IN
**           FIELD 5.3 AND 5.4)
**
**     ALL BYTE ORDERING IS LITTLE-ENDIAN
*/

struct TGAHeader
{
    unsigned char id_field_length;
    unsigned char color_map_type;
    unsigned char image_type_code;
    unsigned short color_map_origin;
    unsigned short color_map_length;
    unsigned char color_map_entry_size;
    unsigned short x_origin;
    unsigned short y_origin;
    unsigned short width;
    unsigned short height;
    unsigned char bits_per_pixel;
    unsigned char image_descriptor;
};
```

```
enum TGAImageTypeCode
{
   TGA_COLOR_MAPPED = 1,
   TGA_TRUE_COLOR = 2,
   TGA_GREY_SCALE = 3
};

class TGA
{
   public:
      TGA ();
      ~TGA ();
      int Load (char *szFilename);
      unsigned Width () { return_tga_header.width; }
      unsigned Height () { return_tga_header. height; }
      void *Image () { return_image; }
      Palette *GetPalette () { return_palette; }
      unsigned ImageTypeCode () { return _tga_header.image_type_code; }
      unsigned BitsPerPixel () { return _tga_header.bits_per_pixel; }
      unsigned BytesPerPixel () { return (_tga_header.bits_per_pixel + 7) / 8; }

   private:
      char *_image;
      unsigned _width;
      unsigned _height;
      TGAHeader _tga_header;
      Palette *_palette;
};

#endif
VESA.H:
#ifndef VESA_H
#define VESA_H

#include "palette.h"
#include "svga.h"
#include "vesavbe.h"
#include "vbeaf.h"

typedef SV_devCtx VESADeviceContext;

extern int VESAInit (unsigned x, unsigned y, unsigned depth_bits, int linear_buffer = 1);
extern void VESAClose ();
extern const VESADeviceContext *VESAGetDeviceContext ();

// All Scitech palettes should have 8 bits per primary

extern void VESALoad (Palette &palette);
extern void VESALine (unsigned x1, unsigned y1, unsigned x2, unsigned y2, unsigned long color);
extern void VESABeginLine ();
extern void VESAEndLine ();
extern void VESALineFast (unsigned x1, unsigned y1, unsigned x2, unsigned y2, unsigned long color);
extern void VESAPoint (unsigned x, unsigned y, unsigned long color);
extern void VESABeginPoint ();
extern void VESAEndPoint ();
extern void VESAPointFast (unsigned x, unsigned y, unsigned long color);
extern void VESAText (char *szText, unsigned x, unsigned y, unsigned long color);
extern void VESAScreenBlt (char *bitmap, unsigned long length);
extern void VESABitBlt (void *bitmap, unsigned width, unsigned height,
    unsigned bytes_per_pixel, unsigned x_offset, unsigned y_offset);
extern void VESAClear (unsigned long color);

#endif
YUV.H:
#ifndef YUV_H
#define YUV_H

/*

Convert R'G'B' (tristimulus values) to Y'UV (color difference
components).

Y'UV is used loosely to mean any color difference coding,
whether it's Y'CBCR, Y'PBPR, or whatever else.

Here, the conversion matrix for Y'PBPR will be used.

To convert from tristimulus values to color difference components:

[ Y'  601 ] [ 0.299 0.587 0.114 ] [ R' ]
[ B'-Y' 601 ]=[-0.299 -0.587 0.886 ]*[ G' ]
[ R'-Y' 601 ] [ -0.701 -0.587 -0.114 ] [ B' ]

Scale PB and PR ranges to [-0.5..+0.5] to get:

[ Y' 601 ] [ 0.299   0.587   0.114  ] [ R' ]
[ PB 601 ]=[-0.168736 -0.331264  0.5  ]*[ G' ]
[ PR 601 ] [ 0.5   -0.418688 -0.081312 ] [ B' ]

The first row, the luma coefficients, sum to 1. The second and
third rows, the chroma coefficients, each sum to zero.

Here is the inverse matrix:

[ R' ] [ 1.0 0.0   1.402  ] [ Y' 601 ]
[ G' ]=[ 1.0 -0.344136 -0.714136 ]*[ PB 601 ]
[ B' ] [ 1.0 1.772   0.0   ] [ PR 601 ]

*/

#include "rgb.h"
#include "palette.h"

#define YUV1055
//#define YUV844
//#define YUV633
//#define YUV422
```

```
#if defined (YUV1055)

// 10:5:5

#define YWIDTH 10
#define UWIDTH 5
#define VWIDTH 5
#define YMASK   0x000FFC00
#define UMASK   0x000003E0
#define VMASK   0x0000001F
#define YSHIFT1 0
#define USHIFT1 0
#define VSHIFT1 3
#define YSHIFT2 12
#define USHIFT2 2
#define VSHIFT2 0

#elif defined (YUV844)

// 8:4:4

#define YWIDTH 8
#define UWIDTH 4
#define VWIDTH 4
#define YMASK   0xFF00
#define UMASK   0x00F0
#define VMASK   0x000F
#define YSHIFT1 0
#define USHIFT1 0
#define VSHIFT1 4
#define YSHIFT2 8
#define USHIFT2 0
#define VSHIFT2 0

#elif defined (YUV633)

// 6:3:3

#define YWIDTH 6
#define UWIDTH 3
#define VWIDTH 3
#define YMASK   0xFC0
#define UMASK   0x038
#define VMASK   0x007
#define YSHIFT1 0
#define USHIFT1 2
#define VSHIFT1 5
#define YSHIFT2 4
#define USHIFT2 0
#define VSHIFT2 0

#else
```

```
// 4:2:2

#define YWIDTH 4
#define UWIDTH 2
#define VWIDTH 2
#define YMASK   0xF0
#define UMASK   0x0C
#define VMASK   0x03
#define YSHIFT1 0
#define USHIFT1 4
#define VSHIFT1 6
#define YSHIFT2 0
#define USHIFT2 0
#define VSHIFT2 0

#endif

struct YUV
{
    unsigned y;
    unsigned u;
    unsigned v;

};

extern void YUVMapInit (Palette &palette);
extern unsigned YUVFindClosestRGB (RGB &rgb, Palette &palette);
extern void RGBtoYUV (RGB &rgb, YUV &yuv);
extern void YUVtoRGB (YUV &yuv, RGB &rgb);
extern void ClipRGB (RGB &rgb, unsigned max);
extern unsigned long PackYUV (YUV &yuv);
extern void UnPackYUV (unsigned long packed_yuv, YUV &yuv);
extern unsigned long YUVDifference (YUV &yuv1, YUV yuv2);

#endif
ARGS.CXX:
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "args.h"
#include "prompt.h"

const char *usage_text =
    "usage: pyramid [profile_frames]\n\n"
    "    profile_frames   Foveate 'profile_frames' number of frames\n"
    "\n"
    "Click the left mouse button to toggle foveation.\n"
    "\n"
    "Press any key to quit the program.\n";

Args::Args (int argc, char **argv)
{
    _filename = new char [256];
```

```
 if (!_filename)
     throw "Args::Args(): memory allocation error";

 strcpy (_filename, "default.tga");

 _profile_frames = 0;
 _half_resolution = 2.5;
 _image_distance = 12 * 2.54;
 _screen_width = 12 * 2.54;

 switch (argc)
 {
    case 2:

    if (argv[1][0] == '/'| argv[1][0] == '-'| argv[1][0] == '?')
        throw usage_text;

    _profile_frames = atoi (argv[1]);

    if (!_profile_frames)
        throw usage_text;

    // Fall through to next case

    case 1;
    break;

    default:
    throw usage_text;
}

// Prompt the user for parameters

printf ("\nFilename [%s]:\t", _filename);
GetString (_filename);

printf ("Half resolution [%0.1f]:\t", _half_resolution);
GetFloat (_half_resolution);

printf ("Screen distance [%d]:\t", _image_distance);
GetUnsigned (_image_distance);

printf ("Screen width [%d]:\t", _screen_width);
GetUnsigned (_screen_width);

// Add a .TGA to the file if it needs it

if ((strlen (_filename) < 5) |
    (strcmpi (&_filename[strlen (_filename) - 4], ".TGA")))
    strcat (_filename, ".TGA");

FILE *fp = fopen (_filename, "r");
```

```
    if (!fp)
    {
        // If the file is not in the current directory, and
        // the environment variable "IMAGE_DIR" exists, then
        // prepend it to the filename.

        char *ev = getenv ("IMAGE_DIR");

        if (ev)
        {
            char *temp = new char [256];

            if (!temp)
                throw "Args::Args(): memory allocation error";

            strcpy (temp, _filename);
            strcpy (_filename, ev);

            // Add a pathname separator if needed

            if (_filename[strlen (_filename) - 1] != '\\'&&
                _filename[strlen (_filename) - 1] != '/')
                strcat (_filename, "\\");

            strcat (_filename, temp);

            fp = fopen (_filename, "r");

            // If you still can't find the file, copy original name back

            if (!fp)
                strcpy (_filename, temp);
            else
                fclose (fp);

            delete [] temp;
        }
    }
    else
    {
        fclose (fp);
    }
}

Args::~Args ()
{
    delete [] _filename;
}

void Args::Dump (unsigned width_pixels)
{
    printf ("%18s %0.1f degree%s\n",
        "Half resolution:",
        _half_resolution,
            _half_resolution == 1 ? " " : "s");
        printf ("%18s %d cm\n",
            "Image distance:",
            _image_distance);
        printf ("%18s %.2f pixels/cm\n",
            "Image resolution:",
            ((float) width_pixels) / _screen_width);
        printf ("%18s %0.2f' (%0.2f degrees)\n",
            "Foveal pixel arc:",
            atan ((((float) _screen_width) / width_pixels) / _image_distance) * (360 / (2 *asin (1.0))) * 60,
            atan ((((float) _screen_width) / width_pixels) / _image_distance) * (360 / (2 *asin (1.0))));
}
TEST.CXX:
#include <conio.h>
#include <malloc.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#include "args.h"
#include "cursor.h"
#include "mouse.h"
#include "palette.h"
#include "ptimer.h"
#include "pyramid.h"
#include "tga.h"
#include "vesa.h"

void assert2 (int value, char *expr, char *file, unsigned lineno)
{
    if (value)
        return;

    VESAClose ();

    fprintf (stderr, "%s:%d (%s)\n", file, lineno, expr);

    throw "Assertion failed";
}

int main (int argc, char **argv)
{
    try
    {
        Args args (argc, argv);

        // Mouse

        printf ("Initializing mouse...\n");

        if (MouseInit ())
            throw "Mouse driver not installed";
```

```
int left_mouse_button = 0;

// Image

printf ("Loading %s...\n", args.Filename ());

TGA tga;

if (tga.Load (args.Filename ()))
    throw "Could not open specified TGA file";

unsigned image_width = tga.Width ();
unsigned image_height = tga.Height ();

// Graphics

const unsigned SCREEN_WIDTH = 1024;
const unsigned SCREEN_HEIGHT = 768;

if (image_width > SCREEN_WIDTH ‖ image_height > SCREEN_HEIGHT)
    throw "The image is too large to be displayed at this resolution";

if (VESAInit (SCREEN_WIDTH, SCREEN_HEIGHT, tga.BitsPerPixel ()))
    throw "Can't initialize VESA driver";

// Cursor

Cursor cursor;

cursor.SetWidth (26);
cursor.SetColor (255, 0);
cursor.SetMaxX (image_width);
cursor.SetMaxY (image_height);
cursor.SetOffsetX ((SCREEN_WIDTH - image_width) / 2);
cursor.SetOffsetY ((SCREEN_HEIGHT - image_height) / 2);

// Palette

switch (tga.ImageTypeCode ())
{
   case TGA_COLOR_MAPPED:
   case TGA_TRUE_COLOR:
   throw "Only greyscale images are supported";

   case TGA_GREY_SCALE:
   {
       Palette palette; // Default greyscale palette

       VESALoad (palette);
}
break,

default:
        throw "Invalid TGA type code";
    }

    VESABitBlt (tga.Image (),
        image_width,
        image_height,
        tga.BytesPerPixel (),
        SCREEN_WIDTH / 2 - image_width / 2,
        SCREEN_HEIGHT / 2 - image_height / 2);

    // Misc. flags and counters

    int foveation_flag = 1;
    unsigned long total_frames = 0;
    unsigned long total_ticks = 0;
    PentiumTimer pentium_timer;
    unsigned total_kilocycles = 0;
    unsigned long time_hi, timer_lo;

    srand (0);

    // Pyramids

    const unsigned LEVELS = 5;

    int heapchk = _heapchk ();

    if (heapchk != _HEAPOK && heapchk != _HEAPEMPTY)
        throw "Heap error";

    const float CONTRAST_THRESHOLD = 0.01;
    const float DOMINANT_FREQUENCY0 = 24;

    EncodePyramid EP (image_width,
        image_height,
        LEVELS,
        CONTRAST_THRESHOLD,
        args.HalfResolution (),
        DOMINANT_FREQUENCY0,
        args.Distance (),
        SCREEN_WIDTH / args.Width ());
    DecodePyramid DP (image_width,
        image_height,
        LEVELS,
        CONTRAST_THRESHOLD,
        args.HalfResolution (),
        DOMINANT_FREQUENCY0,
        args.Distance (),
        SCREEN_WIDTH / args.Width ());

    Level level_0;

    level_0.image = (char *) tga.Image ();
    level_0.width = image_width;
```

```
level_0.height = image_height;

EP.AllocateLevels (&level_0);
DP.AllocateLevels ();

heapchk = _heapchk ();

if (heapchk != _HEAPOK && heapchk != _HEAPEMPTY)
    throw "Heap error";

for (unsigned level = 0; level < LEVELS - 1; level++)
    EP.Reduce (level);

// Swallow any accidental keystrokes

while (kbhit ())
    getch ();

// Main loop

for (;;)
{
    long mouse_motion;

    if (args.ProfileFrames ())
    {
        if (total_frames == args.ProfileFrames ())
            break;

        mouse_motion = (rand () % image_width) - image_width / 2;
        mouse_motion <<= 16;
        mouse_motion += (rand () % image_height) - image_height / 2;
    }
    else
    {
        unsigned short mouse_status = MouseButtonStatus ();

        if (!left_mouse_button && (mouse_status & MOUSE_LEFT_BUTTON_MASK))
        {
            left_mouse_button = 1;
            foveation_flag = !foveation_flag;
        }

        if (!(mouse_status & MOUSE_LEFT_BUTTON_MASK))
            left_mouse_button = 0;

        mouse_motion = MouseMotion ();
    }

    if (kbhit ())
    {
        // swallow character

        getch ();
    break;
}

if (mouse_motion | left_mouse_button)
{
    // Change has occurred

    cursor.UpdateX ((short) (mouse_motion >> 16));
    cursor.UpdateY ((short) (mouse_motion));

    if (foveation_flag)
    {
        clock_t start_ticks = clock ();
        pentium_timer.Start ();

        // Foveate the image

        EP.ComputeThresholds (cursor.X (), cursor.Y ());
        DP.ComputeThresholds (cursor.X (), cursor.Y ());

        // Encode

        for (level = LEVELS - 1; level > 0; level--)
            EP.Expand (level);

        for (level = 0; level < LEVELS - 1; level++)
            EP.Subtract (level);

        // Decode

        DP.SetLevelN (EP.GetLevelN ());

        for (level = LEVELS - 1; level > 0; level--)
        {
            DP.Expand (level);
            DP.Add (EP.GetLevel (level - 1), level - 1);
        }

        pentium_timer.Stop ();
        pentium_timer.Elapsed (timer_hi, timer_lo);

        total_ticks += clock () - start_ticks;
        ++total_frames;

        if (timer_hi)
            throw "Pentium timer overflow";

        total_kilocycles += time_lo / 1000;

        VESABitBlt (DP.GetLevel (0)->image,
            image_width,
            image_height,
            tga.BytesPerPixel (),
                (SCREEN_WIDTH >> 1)-(image_width >> 1),
                (SCREEN_HEIGHT >> 1)-(image_height >> 1));
            }
            else
            {
                VESABitBlt (tga.Image (),
                    image_width,
                    image_height,
                    tga.BytesPerPixel (),
                    SCREEN_WIDTH / 2 - image_width / 2,
                    SCREEN_HEIGHT / 2 - image_height / 2);
            }

            cursor.Draw (VESALine);
        }
    }

    VESAClose ();

    printf ("%18s %s\n", "Filename:", args.Filename ());
    printf ("%18s %d X %d", "Dimensions:", image_width, image_height);

    switch (tga.ImageTypeCode ())
    {
        case TGA_COLOR_MAPPED:
        printf ("color mapped image\n");
        break;

        case TGA_TRUE_COLOR:
        printf ("true color image\n");
        break;

        case TGA_GREY_SCALE:
        printf ("greyscale image\n");
        break;

        default:
        printf ("(unknown image type code)\n");
    }

    args.Dump (SCREEN_WIDTH);

    if (total_frames != 0)
    {
        printf ("%18s %d\n",
            "Total frames:",
            total_frames);
        printf ("%18s %.2f seconds\n",
            "Total time:",
            ((float) total_ticks) / CLOCKS_PER_SEC);
        printf ("%18s %0.2f frames/sec\n",
            " ",
            ((float) total_frames) / (((float) total_ticks) / CLOCKS_PER_SEC));
        printf ("%18s %0.0f pixels/sec\n",
                " ",
                ((float) total_frames * image_width * image_height / (((float) total_ticks) / CLOCKS_PER_SEC));
            printf ("%18s %0.0f megacycles\n",
                "Pentium Cycles:",
                ((float) total_kilocycles) / 1000);
            printf ("%18s %0.0f megacycles/frame\n",
                " ",
                (((float) total_kilocycles) / 1000) / total_frames);
            printf ("%18s %0.1f cycles/pixel\n\n",
                " ",
                ((float) total_kilocycles) / total_frames / image_width / image_height * 1000);
        }

        return 0;
    }
    catch (const char *msg)
    {
        VESAClose ();

        fprintf (stderr, "%s\n\n", msg);

        return -1;
    }
}
ARGS.H:
#ifndef ARGS_H
#define ARGS_H

class Args
{
    public:
        Args (int argc, char **argv);
        ~Args ();
        char *Filename () { return_filename; }
        float HalfResolution () { return_half_resolution; }
        unsigned Distance () { return_image_distance; }
        unsigned Width () { return_screen_width; }
        unsigned ProfileFrames () { return_profile_frames; }
        void Dump (unsigned width_pixels);

    private:
        char *_filename;
        float _half_resolution;
        unsigned _image_distance;
        unsigned _screen_width;
        unsigned _profile_frames;
};

#endif
ARGS.CXX:
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
#include "args.h"
#include "prompt.h"

const char *usage_text =
    "usage: superpix [profile_frames]\n\n"
    "   profile_frames   Foveate 'profile_frames' number of frames\n"
    "\n"
    "Click the left mouse button to toggle foveation.\n"
    "\n"
    "Press any key to quit the program.\n";

Args::Args (int argc, char **argv)
{
    _filename = new char [256];

    if (!_filename)
        throw "Args::Args(): memory allocation error";

    strcpy (_filename, "default.tga");

    _profile_frames = 0;
    _half_resolution = 2.5;
    _image_distance = 12 * 2.54;
    _w0 = 1;
    _screen_width = 12 * 2.54;

    switch (argc)
    {
        case 2:

        if (argv[1][0] == '/' | argv[1][0] == '-' | argv[1][0] == '?')
            throw usage_text

        _profile_frames = atoi (argv[1]);

        if (!_profile_frames)
            throw usage_text;

        // Fall through to next case

        case 1:
        break;

        default:
        throw usage_text;
    }

    // Prompt the user for parameters

    printf ("\nFilename [%s]:\t", _filename);
    GetString (_filename);

    printf ("Half resolution [%0.1f]:\t", _half_resolution);
```

```
GetFloat (_half_resolution);

printf ("Foveal pixel size [%d]:\t", _w0);
GetUnsigned (_w0);

printf ("Screen distance [%d]:\t", _image_distance);
GetUnsigned (_image_distance);

printf ("Screen width [%d]:\t", _screen_width);
GetUnsigned (_screen_width);

// Add a .TGA to the file if it needs it

if ((strlen (_filename) < 5) |
    (strcmpi (&_filename[strlen (_filename) - 4], ".TGA")))
    strcat (_filename, ".TGA");

FILE *fp = fopen (_filename, "r");

if (!fp)
{
    // If the file is not in the current directory, and
    // the environment variable "IMAGE_DIR" exists, then
    // prepend it to the filename.

    char *ev = getenv ("IMAGE_DIR");

    if (ev)
    {
        char *temp = new char [256];

        if (!temp)
            throw "Args::Args(): memory allocation error";

        strcpy (temp, _filename);
        strcpy (_filename, ev);

        // Add a pathname separator if needed

        if (_filename[strlen (_filename) - 1] != '\\' &&
            _filename[strlen (_filename) - 1] != '/')
            strcat (_filename, "\\");

        strcat (_filename, temp);

        fp = fopen (_filename, "r");

        // If you still can't find the file, copy original name back

        if (!fp)
            strcpy (_filename, temp);
        else
            fclose (fp);
            delete [] temp;
         }
      }
      else
      {
         fclose (fp);
      }
}

Args::~Args ()
{
   delete [] _filename;
}

void Args::Dump (unsigned width_pixels)
{
   printf ("%18s %0.1f degrees%s\n",
      "Half resolution:",
      _half_resolution,
      _half_resolution == 1 ? " " : "s");
   printf ("%18s %d pixel%s\n",
      "Foveal pixel size:",
      _w0,
      _w0 == 1 ? " " : "s");
   printf ("%18s %d cm\n",
      "Image distance:",
      _image_distance);
   printf ("%18s %.2f pixels/cm\n",
      "Image resolution:",
      ((float) width_pixels) / _screen_width);
   printf ("%18s %0.2f' (%0.2f degrees)\n",
      "Foveal pixel arc:",
      atan ((((float) _screen_width) / width_pixels) / _image_distance) * (360 / 2 * asin (1.0))) * 60,
      atan ((((float) _screen_width) / width_pixels) / _image_distance) * (360 / 2 * asin (1.0))));
}
TEST.CXX:
#include <assert.h>
#include <conio.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "args.h"
#include "cursor.h"
#include "foveator.h"
#include "integrat.h"
#include "interpol.h"
#include "mouse.h"
#include "palette.h"
#include "ptimer.h"
#include "tga.h"
#include "vesa.h"
#include "yuv.h"
```

```
#include "ztimer.h"

int main (int argc, char **argv)
{
   try
   {
      Args args (argc, argv);

      // Mouse

      printf ("Initializing mouse...\n");

      if (MouseInit ())
          throw "Mouse driver not installed";

      int left_mouse_button = 0;

      // Image

      printf ("Loading %s...\n", args.Filename ());

      TGA tga;

      if (tga.Load (args.Filename ()))
          throw "Could not open specified TGA file";

      unsigned image_width = tga.Width ();
      unsigned image_height = tga.Height ();

      // Graphics

      const unsigned SCREEN_WIDTH = 1024;
      const unsigned SCREEN_HEIGHT = 768;

      if (image_width > SCREEN_WIDTH | image_height > SCREEN_HEIGHT)
          throw "The image is too large to be displayed at this resolution";

      if (VESAInit (SCREEN_WIDTH, SCREEN_HEIGHT, tga.BitsPerPixel ()))
          throw "Can't initialize VESA driver";

      // Cursor

      Cursor cursor;

      cursor.SetWidth (26);
      cursor.SetColor (255, 0);
      cursor.SetMaxX (image_width);
      cursor.SetMaxY (image_height);
      cursor.SetOffsetX ((SCREEN_WIDTH - image_width) / 2);
      cursor.SetOffsetY ((SCREEN_HEIGHT - image_height) / 2);

      // Palette

      Palette *palette;
```

```
switch (tga.ImageTypeCode ())
{
    case TGA_COLOR_MAPPED:

    palette = tga.GetPalette ();

    VESALoad (*palette);
    YUVMapInit (*palette);

    break;

    case TGA_TRUE_COLOR:
    cursor.SetColor (0x00FFFFFF, 0x00000000);
    break;

    case TGA_GREY_SCALE:
    {
        Palette palette; // Default greyscale palette

        VESALoad (palette);
    }
    break;

    default:
    throw "Invalid type code";
}

VESABitBlt (tga.Image (),
    image_width,
    image_height,
    tga.BytesPerPixel (),
    SCREEN_WIDTH / 2 - image_width / 2,
    SCREEN_HEIGHT / 2 - image_height / 2);

// Foveator

Foveator foveator;
foveator.SetScreenResolution (image_width, image_height);
foveator.SetHalfResolution (args.HalfResolution ());
foveator.SetDistance (args.Distance ());
foveator.SetPixelsPerCM (SCREEN_WIDTH / args.Width ());
foveator.SetW0 (args.W0 ());

unsigned super_pixel_count;
SuperPixel *super_pixels;

char *foveated_image = new char [image_width * image_height * tga.BytesPerPixel ()];

if (!foveated_image)
    throw "Memory allocation error";

// Misc. flags and counters
```

```
int foveation_flag = 1;
unsigned long total_frames = 0;
unsigned long total_ticks = 0;
unsigned long total_super_pixels = 0;

ZTimerInit ();

srand (0);

PentiumTimer pentium_timer;
unsigned total_kilocycles = 0;
unsigned long timer_hi, timer_lo;

// Swallow any accidental keystrokes

while (kbhit ())
    getch ();

// Main loop

for (;;)
{
    long mouse_motion;

    if (args.ProfileFrames ())
    {
        if (total_frames == args.ProfileFrames ())
            break;

        mouse_motion = (rand () % image_width) - image_width / 2;
        mouse_motion <<= 16;
        mouse_motion += (rand () % image_height) - image_height / 2;
    }
    else
    {
        unsigned short mouse_status = MouseButtonStatus ();

        if (!left_mouse_button && (mouse_status & MOUSE_LEFT_BUTTON_MASK))
        {
            left_mouse_button = 1;
            foveation_flag = !foveation_flag;
        }

        if (!(mouse_status & MOUSE_LEFT_BUTTON_MASK))
            left_mouse_button = 0;

        mouse_motion = MouseMotion ();
    }

    if (kbhit ())
    {
        // swallow character

        getch ();
        break;
    }

    if (mouse_motion | left_mouse_button)
    {
        // Change has occurred

        cursor.UpdateX ((short)(mouse_motion >> 16));
        cursor.UpdateY ((short)(mouse_motion));

        if (foveation_flag)
        {
            LZTimerOn ();
            pentium_timer.Start ();

            super_pixels = foveator.Foveate (cursor.X (),
                cursor.Y (),
                super_pixel_count);

            total_super_pixels += super_pixel_count;

            Integrate (super_pixels,
                super_pixel_count,
                tga.Image (),
                image_width,
                palette,
                tga.ImageTypeCode ());

            Interpolate (super_pixels,
                super_pixel_count,
                foveated_image,
                image_width,
                tga.ImageTypeCode ());

            LZTimerOff ();
            pentium_timer.Stop ();
            pentium_timer.Elapsed (timer_hi, timer_lo);
            total_ticks += LZTimerCount ();
            ++total_frames;

            if (timer_hi)
                throw "Pentium timer overflow";

            total_kilocycles += timer_lo / 1000;

            VESABitBlt (foveated_image,
                image_width,
                image_height,
                tga.BytesPerPixel (),
                (SCREEN_WIDTH >> 1) - (image_width >> 1),
                (SCREEN_HEIGHT >> 1) - (image_height >> 1));
        }
        else
```

```
            {
                VESABitBlt (tga.Image (),
                    image_width,
                    image_height,
                    tga.BytesPerPixel (),
                    SCREEN_WIDTH / 2 - image_width / 2,
                    SCREEN_HEIGHT / 2 - image_height / 2);
            }

            cursor.Draw (VESALine);
        }
    }

    VESAClose ();

    printf ("%17s: %s\n", "Filename", args.Filename ());
    printf ("%17s: %d X %d ", "Dimensions", image_width, image_height);

    switch (tga.ImageTypeCode ())
    {
        case TGA_COLOR_MAPPED:
        printf ("color mapped image\n");
        break;

        case TGA_TRUE_COLOR:
        printf ("true color image\n");
        break;

        case TGA_GREY_SCALE:
        printf ("greyscale image\n");
        break;

        default:
        printf ("(unknown image type code)\n");
    }

    printf ("%17s: %0.1f degree%s\n",
        "Half resolution",
        args.HalfResolution (),
        args.HalfResolution () == 1 ? " " : "s");
    printf ("%17s: %d pixel%s\n",
        "Foveal pixel size",
        args.W0 (),
        args.WO () == 1 ? " " : "s");
    printf ("%17s: %d cm\n",
        "Image distance",
        args.Distance ());
    printf ("%17s: %.2f pixels/cm\n",
        "Image resolution",
        ((float) SCREEN_WIDTH) / args.Width ());
    print ("%17s: %0.2f' (%0.2f degrees)\n",
        "Foveal pixel arc",
        atan ((((float) args.Width ()) / SCREEN_WIDTH) / args.Distance ()) * (360 / (2 * asin (1.0))) * 60,
        atan ((((float) args.Width ()) / SCREEN_WIDTH) / args.Distance ()) * (360 / (2 * asin (1.0))));
        if (total_frames != 0)
        {
            printf ("%17s: %d\n",
                "Total frames",
                total_frames);
            printf ("%17s: %.2f seconds\n",
                "Total time",
                (double) total_ticks * LZTIMER_RES);
            printf ("%17s  %0.2f frames/sec\n",
                " ",
                ((float) total_frames) / LZTIMER_RES / total_ticks);
            printf ("%17s  %0.0f pixels/sec\n",
                " ",
                ((float) total_frames * image_width * image_height) / LZTIMER_RES / total_ticks);
            printf ("%17s: %d sp/frame\n",
                "Avg superpixels",
                total_super_pixels / total_frames);

            printf ("%17s: %d:1\n",
                "Compression",
                (image_width * image_height) / (total_super_pixels / total_frames));
            printf ("%17s: %0.0f megacycles\n",
                "Pentium Cycles",
                ((float) total_kilocycles) / 1000);
            printf ("%17s  %0.1f megacycles/frame\n",
                " ",
                (((float) total_kilocycles) / 1000) / total_frames);
            printf ("%17s  %0.1f cycles/pixel\n",
                " ",
                ((float) total_kilocycles) / total_frames / image_width / image_height * 1000);
        }

        delete [] foveated_image;

        return 0;
    }
    catch (const char *msg)
    {
        VESAClose ();

        fprintf (stderr, "%s\n", msg);

        return -1;
    }
}

ARGS.H:
#ifndef ARGS_H
#define ARGS_H

class Args
{
    public:
        Args (int argc, char **argv);
        ~Args ();
        char *Filename () { return _filename; }
        unsigned W0 () { return _w0; }
        float HalfResolution () { return _half_resolution; }
        unsigned Distance () { return _image_distance; }
        unsigned Width () { return _screen_width; }
        unsigned ProfileFrames () { return _profile_frames; }
        void Dump (unsigned width_pixels);

    private:
        char *_filename;
        unsigned _w0;
        float _half_resolution;
        unsigned _image_distance;
        unsigned _screen_width;
        unsigned _profile_frames;
};

#endif
```

CLAIMS

We claim:

1. A system for conveyance of video image data from a first computer system to a second computer system, the system comprising:

   a first computer system for real-time foveated transformational coding of video image data;

   a second computer system for decoding the foveated transformationally coded video image data; and

   a communications channel linking the first computer system and the second computer system.

2. The system of claim 1 in which the real-time foveated transformational coding of video image data includes manipulation of a subset of the video image data selected by a first foveation function.

3. The system of claim 2 in which the first foveation function corresponds to a first human perception response to first image data derived from the video image data.

4. The system of claim 2 in which the first foveation function corresponds to a first human perception spatial frequency response to the first image data and a first human perception eccentricity visual response to the first image data.

5. The system of claim 3 or 4 in which the first foveation function is a function of a current gaze point.

6. The system of claim 1 in which the real-time foveated transformational coding of the video image data includes threshold filtering of second image data derived from the video image data.

7. The system of claim 6 in which the threshold filtering of the second image data is based upon a second foveation function.

8. The system of claim 7 in which the second foveation function corresponds to a second human perception response to the second image data.

9. The system of claim 7 in which the second foveation function corresponds to a second human perception spatial frequency response to the second image data and a second human perception eccentricity visual response function to the second image data.

10. The system of claim 8 or 9 in which the second foveation function is based upon a current gaze point.

11. A system for conveyance of video information from a first computer system to a second computer system, the system comprising:

a first computer system for foveated transformational coding of video image data;

a second computer system for decoding the foveated transformationally coded video image data;

a communications channel linking the first computer system and the second computer system; and

wherein the foveated transformational coding of the video image data consists of foveated Laplacian pyramidal transformational coding, foveated wavelet pyramidal transformational coding, or foveated block discrete cosine transformational coding.

12. The system of claim 11 in which the foveated transformational coding of the video image data is performed in real-time.

13. The system of claim 11 in which the foveated tranformational coding of the video image data includes manipulation of a foveated subset of the video image data selected by a foveation function.

14. The system of claim 13 in which the foveation function is a function of a current gaze point.

15. The system of claim 13 in which the foveated transformational coding of the video image data includes threshold filtering of image data derived from the video image data.

16. The system of claim 15 in which the threshold filtering of the derived image data corresponds to a human perception response to the derived image data.

17. The system of claim 15 in which the threshold filtering is based upon a human perception spatial frequency response function to the image derived from the video image data and a human perception eccentricity visual response function to the image derived from the video image data.

18. A method for compression of video image data comprising the steps of:

acquiring digitized video image data; and

performing by a computer system, real-time foveated transformational coding of the digitized video image data.

19. The method of claim 18 in which the real-time foveated transformational coding of the digitized video image data includes manipulation of a subset of the digitized video image data selected by a first foveation function.

20. The method of claim 19 in which the first foveation function corresponds to a first human perception response to first image data derived from the digitized video image data.

21. The method of claim 19 in which the first foveation function corresponds to a first human perception spatial frequency response to the first image data and a first human perception eccentricity visual response to the first image data.

22. The method of claim 20 or 21 in which the first foveation function is a function of a current gaze point.

23. The method of claim 18 in which the real-time foveated transformational coding of the video image data includes threshold filtering of second image data derived from the video image data.

24. The method of claim 23 in which the threshold filtering of the second image data is based upon a second foveation function.

25. The method of claim 24 in which the second foveation function corresponds to a second human perception response to the second image data.

26. The system of claim 24 in which the second foveation function corresponds to a second human perception spatial frequency response to the second image data and a second human perception eccentricity visual response function to the second image data.

27. The system of claim 25 or 26 in which the second foveation function is based upon a current gaze point.

28. A method for compression of video image data comprising the steps of:

acquiring digitized video image data; and

performing by a computer system, foveated transformational coding of the digitized video image data; wherein the foveated transformational coding of the digitized video image data consists of foveated Laplacian pyramidal transformational coding, foveated wavelet pyramidal transformational coding, or foveated block discrete cosine transformational coding.

29. The method of claim 28 in which the step of performing foveated transformational coding of the digitized video data is performed in real-time.

30. The method of claim 28 in which the foveated Laplacian pyramidal transformational coding of the digitized video image data comprises the steps of:

reducing the digitized video image data;

expanding a foveated subset of the reduced digitized video image data; and

subtracting the expanded foveated subset of the reduced digitized video image data to obtain difference video image data.

31. The method of claim 30 in which the foveated subset of the reduced digitized video image data is determined by a first foveation function that correlates to a first human perception spatial frequency response function to the reduced video image data image and a first human perception eccentricity visual response function to the reduced video image data.

32. The method of claim 31 in which the first foveation function is a function of a current gaze point.

33. The method of claim 31 further comprising the steps of:

thresholding the difference video image data to obtain thresholded difference video image data; and

quantizing the thresholded difference video image data to obtain quantized thresholded difference video image data.

34. The method of claim 33 further comprising the step of compressing the quantized thresholded difference video image data.

35. The method of claim 33 in which the step of thresholding the difference video image data is determined by a second foveation function that correlates to a second human perception spatial frequency response function to the difference video image data image and a second human perception eccentricity visual response function to the difference video image data.

36. The method of claim 35 in which the second foveation function is a function of the current gaze point.

37. The method of claim 35 in which the first foveation function and the second foveation function are identical.

38. The method of claim 36 in which the first foveation function and the second foveation function are identical.

39. The method of claim 28 in which the foveated block discrete cosine transformational coding of the digitized video image data comprises the steps of:

dividing the digitized video image data into non-overlapping blocks;

applying a discrete cosine transform to each block to obtain transformed video image data;

thresholding the transformed video image data with a foveation function to result in thresholded video image data; and quantizing the thresholded video image data.

40. The method of claim 39 in which the foveation function correlates to a human perception spatial frequency response function to the transformed video image data and a human perception eccentricity visual response function to the transformed video image data.

41. The method of claim 39 in which the foveation function is a function of a current gaze point.

42. The method of claim 39 further comprising the step of compressing the quantized thresholded difference video image data.

43. The method of claim 28 in which the foveated wavelet pyramidal transformational coding of the digitized video image data comprises the steps of:

convolving with a plurality of filtering kernals over the digitized video image to obtain plural subband images, the convolutions comprising high-pass convolution over a foveated subset of the digitized video image, the foveated subset selected by a first foveation function; and

thresholding the plural subband images with a second foevation function to obtain plural thresholded subband images.

44. The method of claim 43 further comprising the steps of:

quantizing the plural thresholded subband images.

45. A method of compressing a recorded video sequence, the method comprising the steps of:

exposing an observer to the recorded video sequence;

measuring periodically where the observer fixates when exposed to the recorded video sequence to obtain a likely gaze point for each period;

coding the recorded video sequence according to a foveation transformational coding employing the likely gaze point in each period.

46. A method of compressing a recorded video sequence, the method comprising the steps of:

exposing a group of observers to the recorded video sequence;

measuring periodically where the observers fixate when exposed to the recorded video sequence to obtain a statistical likely gaze point for each period; and

coding the recorded video sequence according to a transformational coding employing the statistical likely gaze point for each period.

47. A method of displaying a compressed video sequence comprising the step of displaying a recorded video sequence compressed according to the method of claim 45.

48. A method of displaying a compressed video sequence comprising the step of displaying a recorded video sequence compressed according to the method of claim 46.

49. A method of compressing a recorded video image, the method comprising the steps of:

exposing an observer to the recorded video image;

measuring where the observer fixates when exposed to the recorded video image to obtain a likely gaze point;

coding the recorded video image according to a foveation transformational coding employing the likely gaze point as a foveation function.

FIG. 1

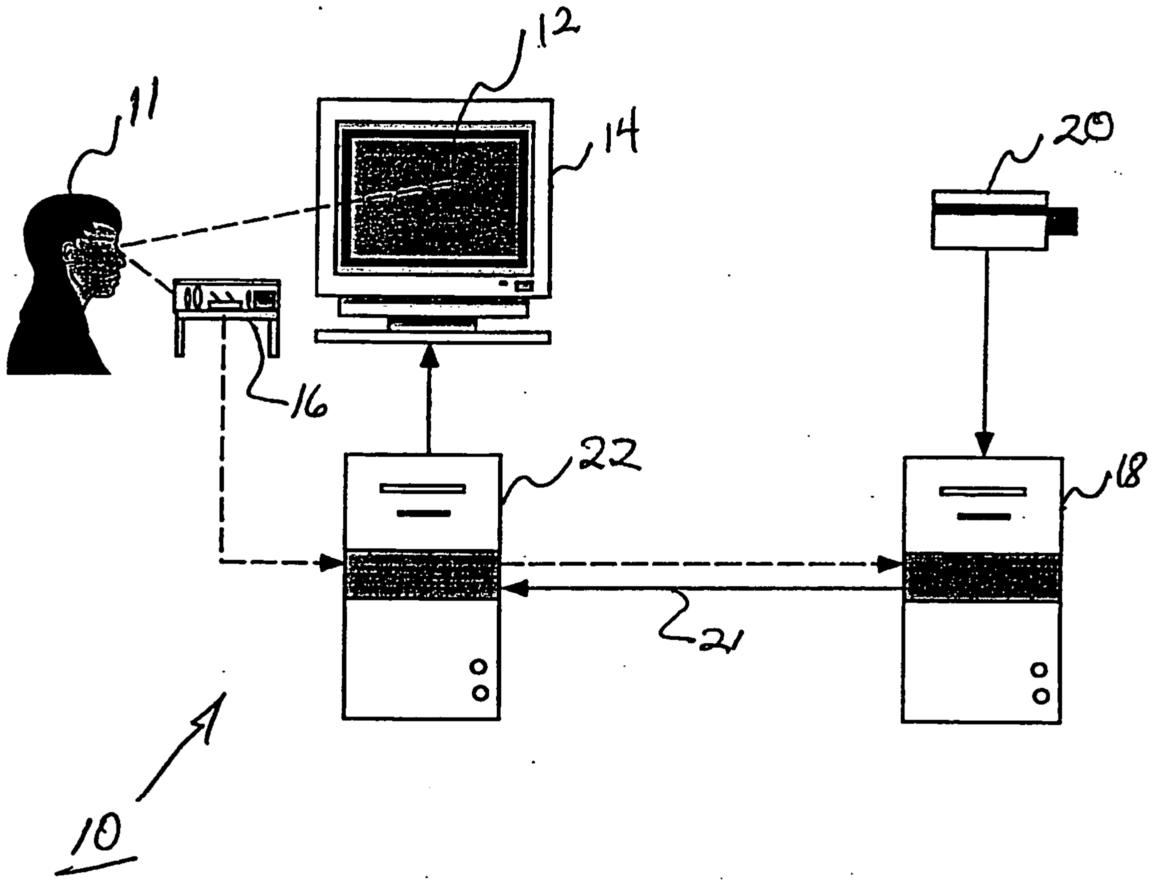


FIG. 2

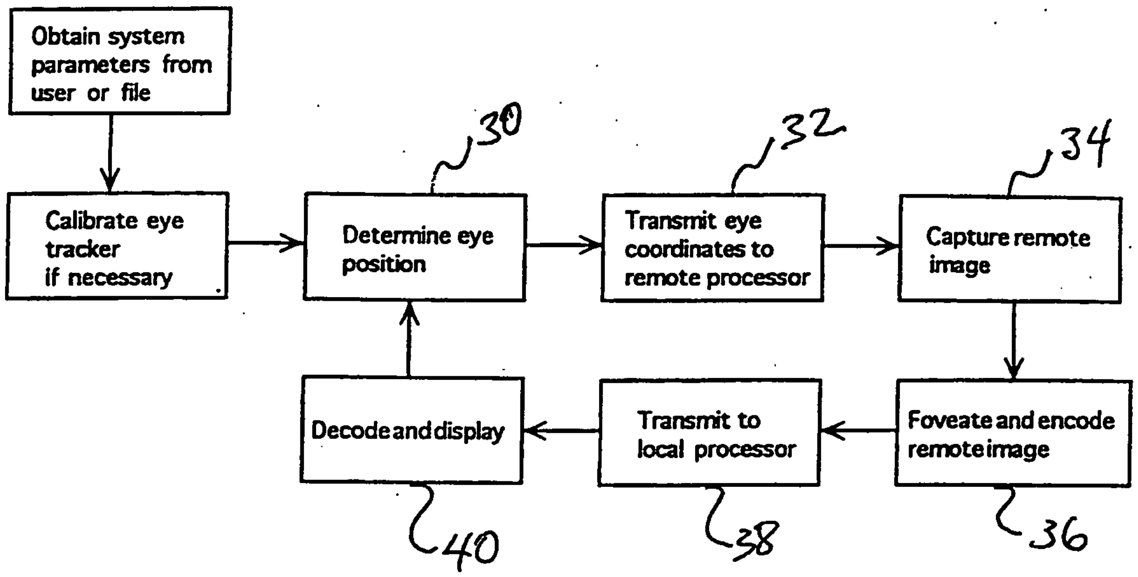

Obtain system parameters from user or file
Calibrate eye tracker if necessary
Determine eye position
30
Transmit eye coordinates to remote processor
32
Capture remote image
34
Foveate and encode remote image
36
Transmit to local processor
38
Decode and display
40

FIG. 3

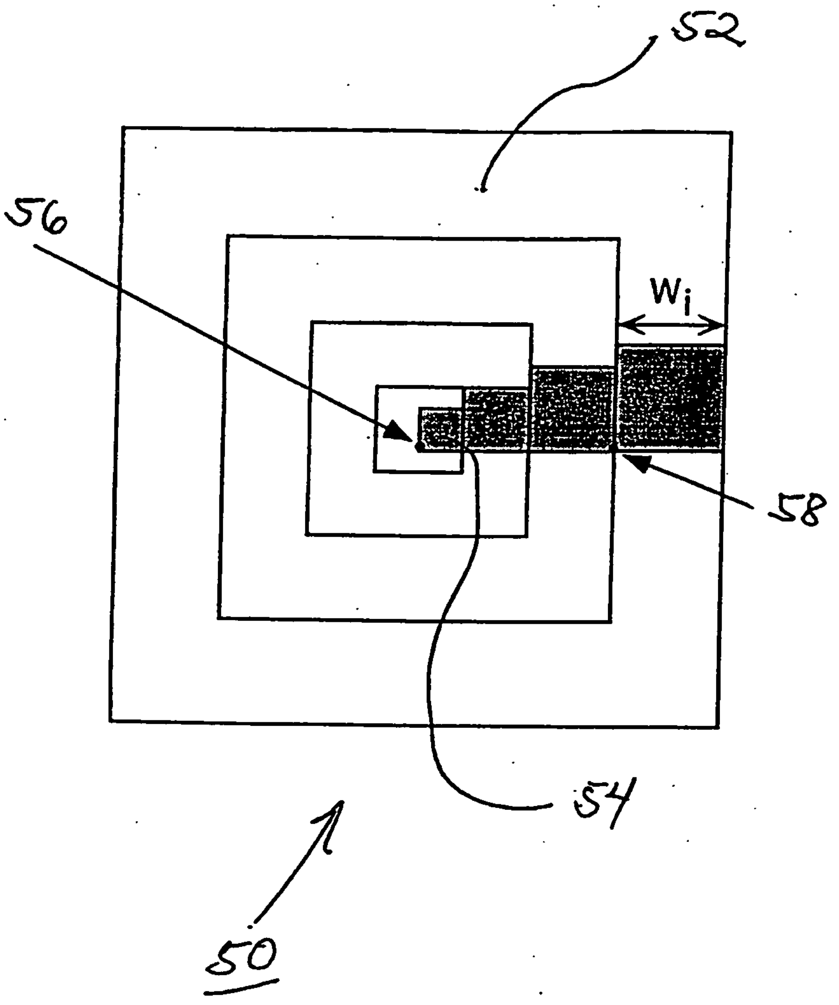


FIG. 4

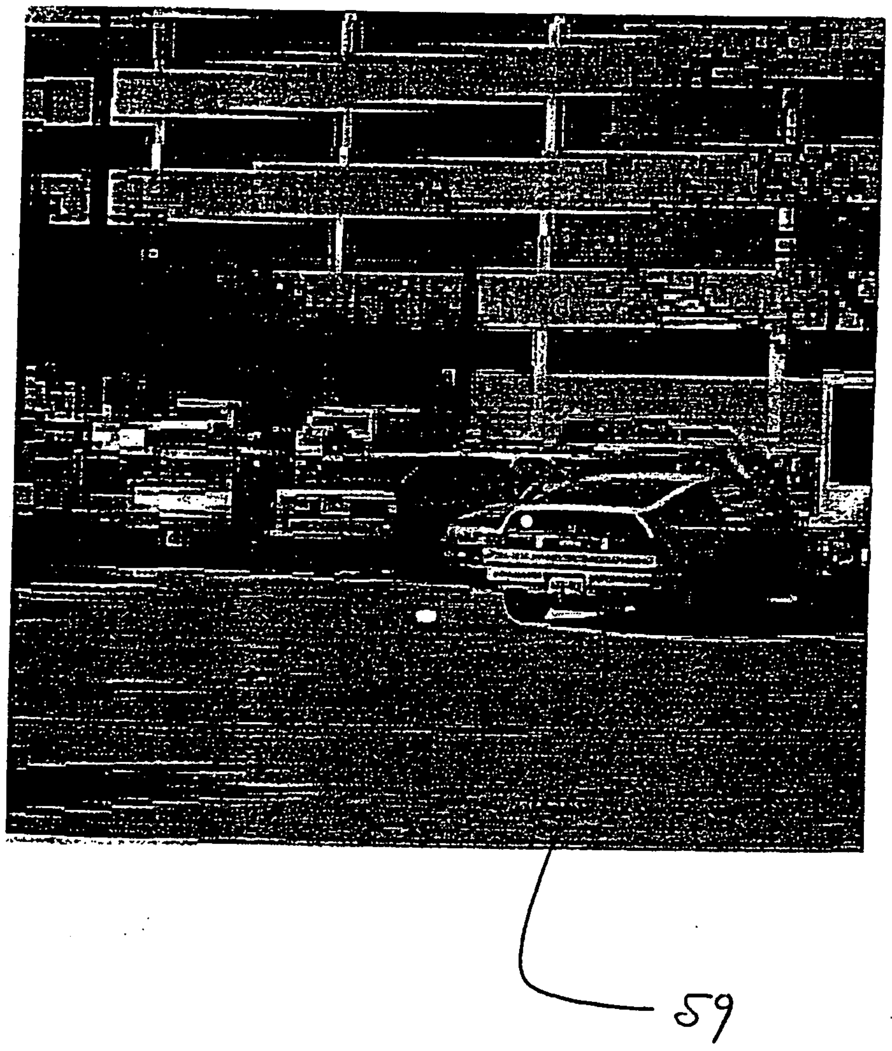


FIG. 5

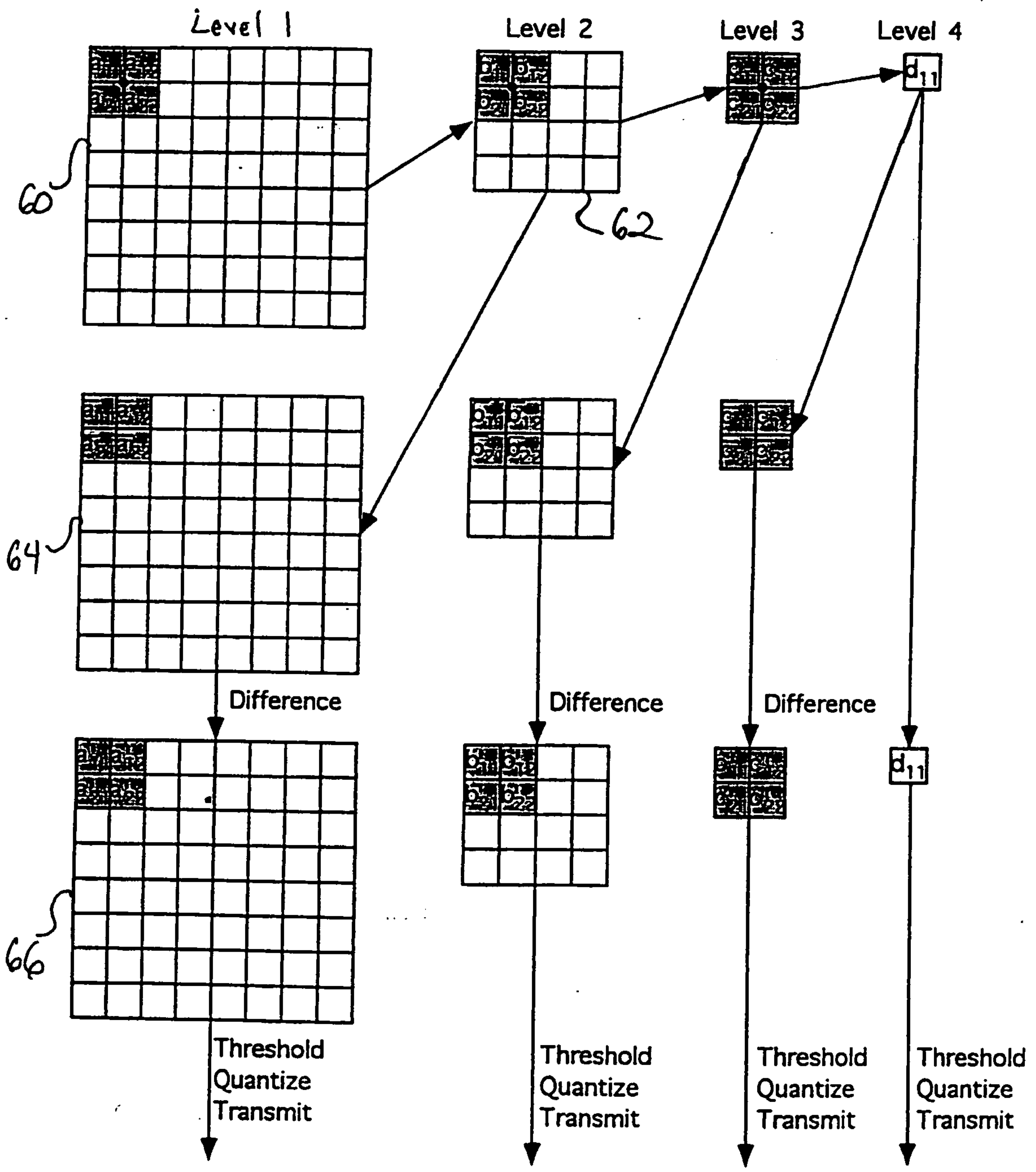

Level 1
Level 2
Level 3
Level 4
d_{11}
60
62
64
66
Difference
Difference
Difference
d_{11}
Threshold
Quantize
Transmit
Threshold
Quantize
Transmit
Threshold
Quantize
Transmit
Threshold
Quantize
Transmit

FIG. 6

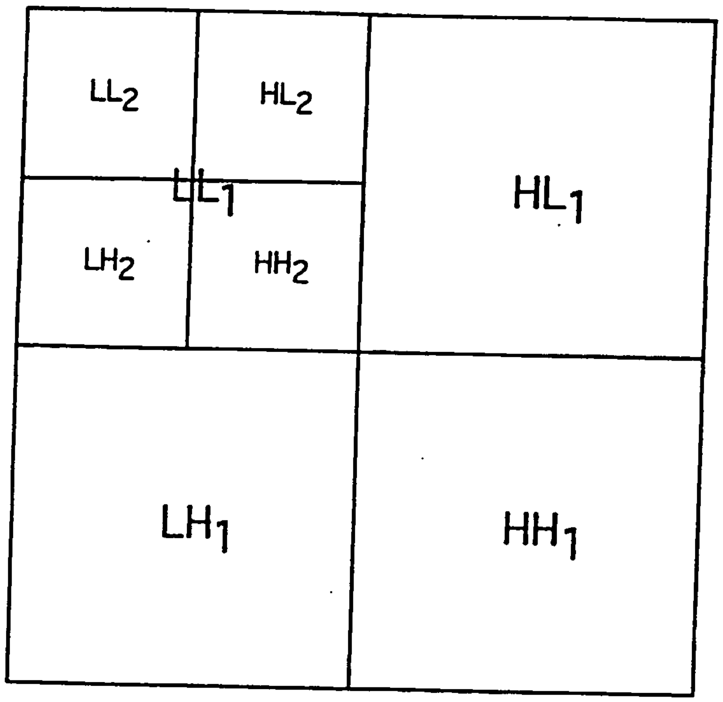


FIG. 7

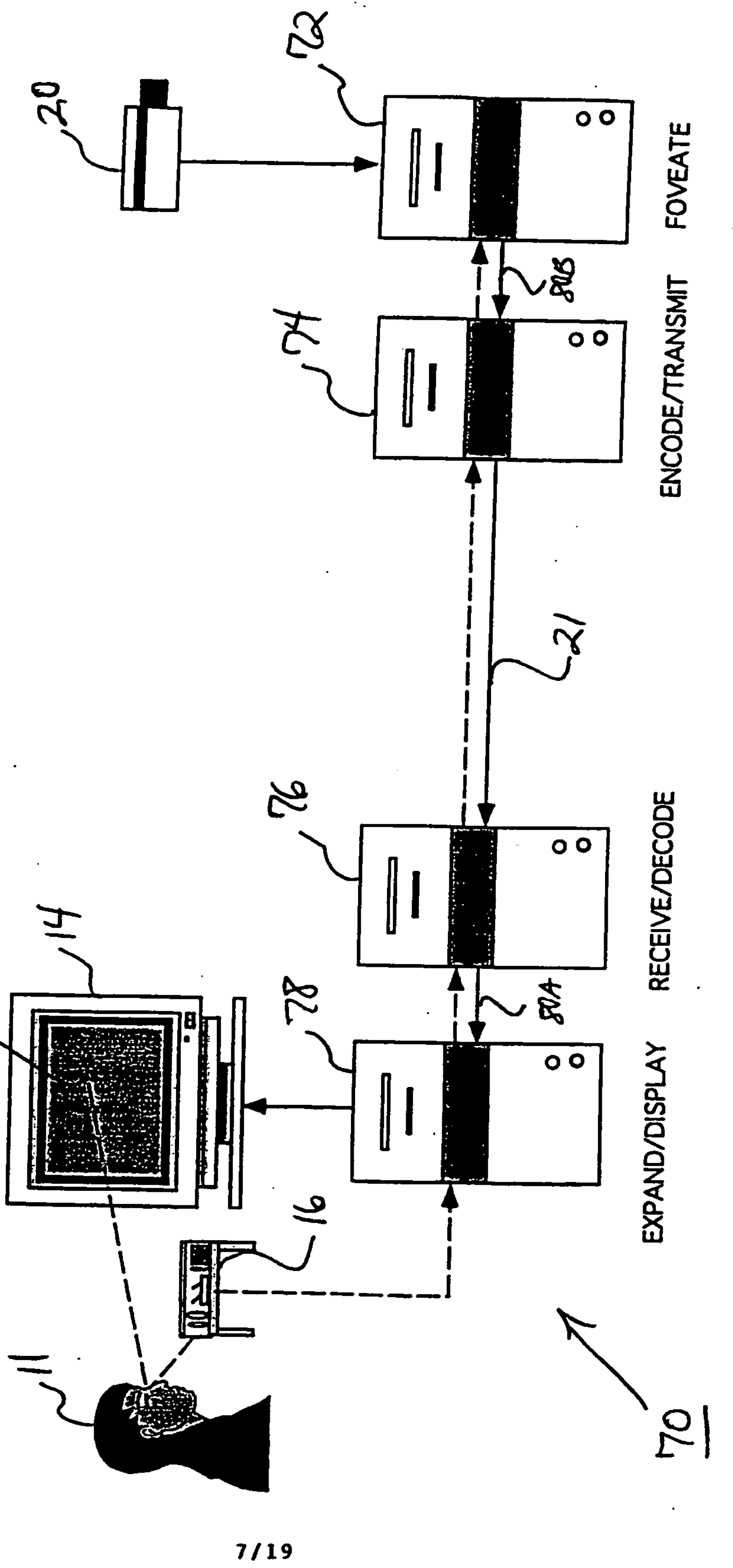

20
72
FOVEATE
88B
74
ENCODE/TRANSMIT
21
76
RECEIVE/DECODE
88A
78
EXPAND/DISPLAY
12
14
16
11
70

FIG. 8

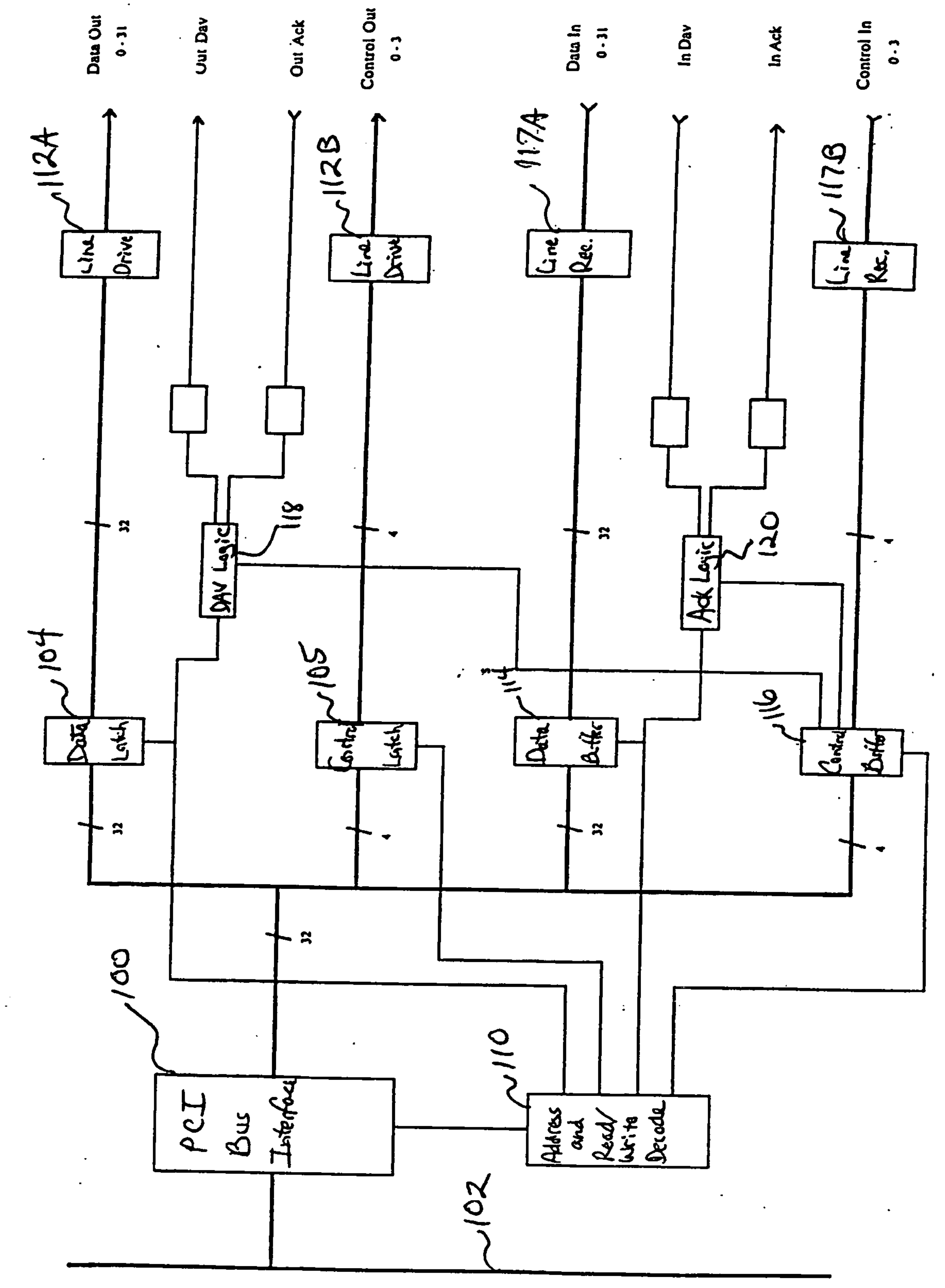

Data Out 0-31
Out Dav
Out Ack
Control Out 0-3
Data In 0-31
In Dav
In Ack
Control In 0-3
112A
Line Drive
112B
Line Drive
117A
Line Rec.
117B
Line Rec.
118
DAV Logic
120
Ack Logic
104
Data Latch
105
Control Latch
114
Data Buffer
116
Control Buffer
32
4
100
PCI Bus Interface
110
Address and Read/Write Decode
102

FIG. 9

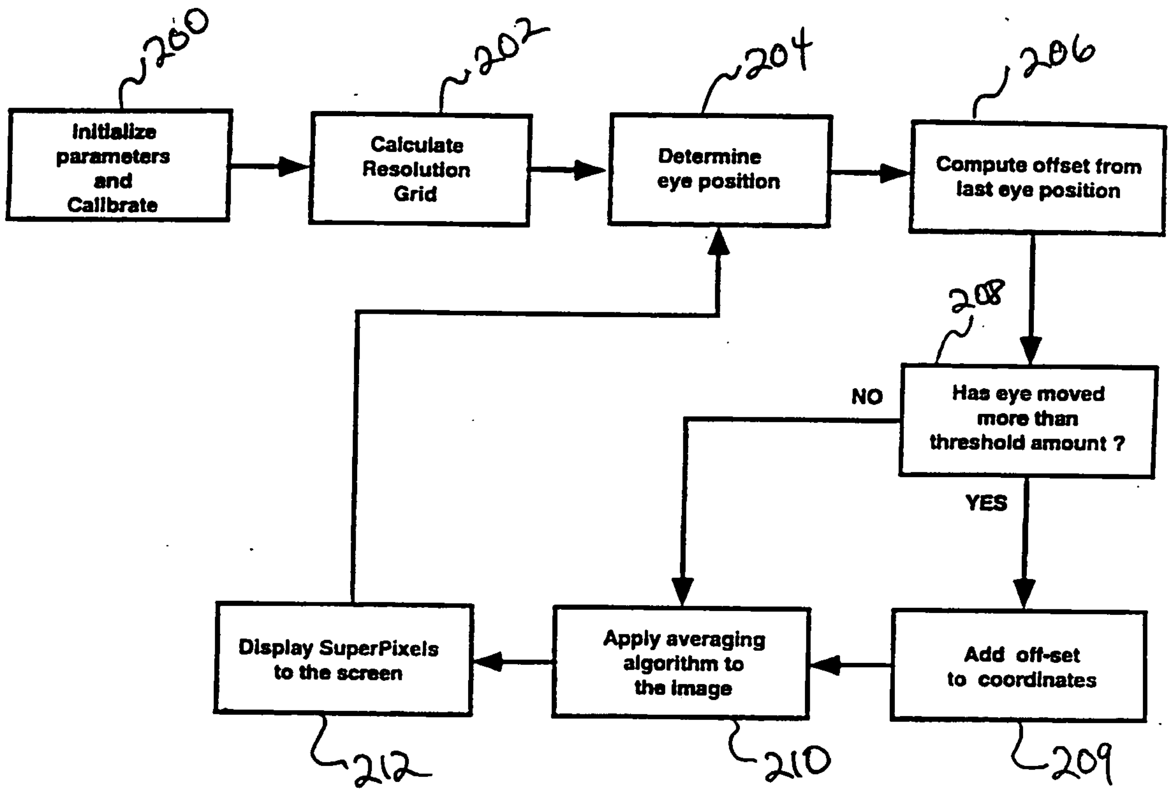
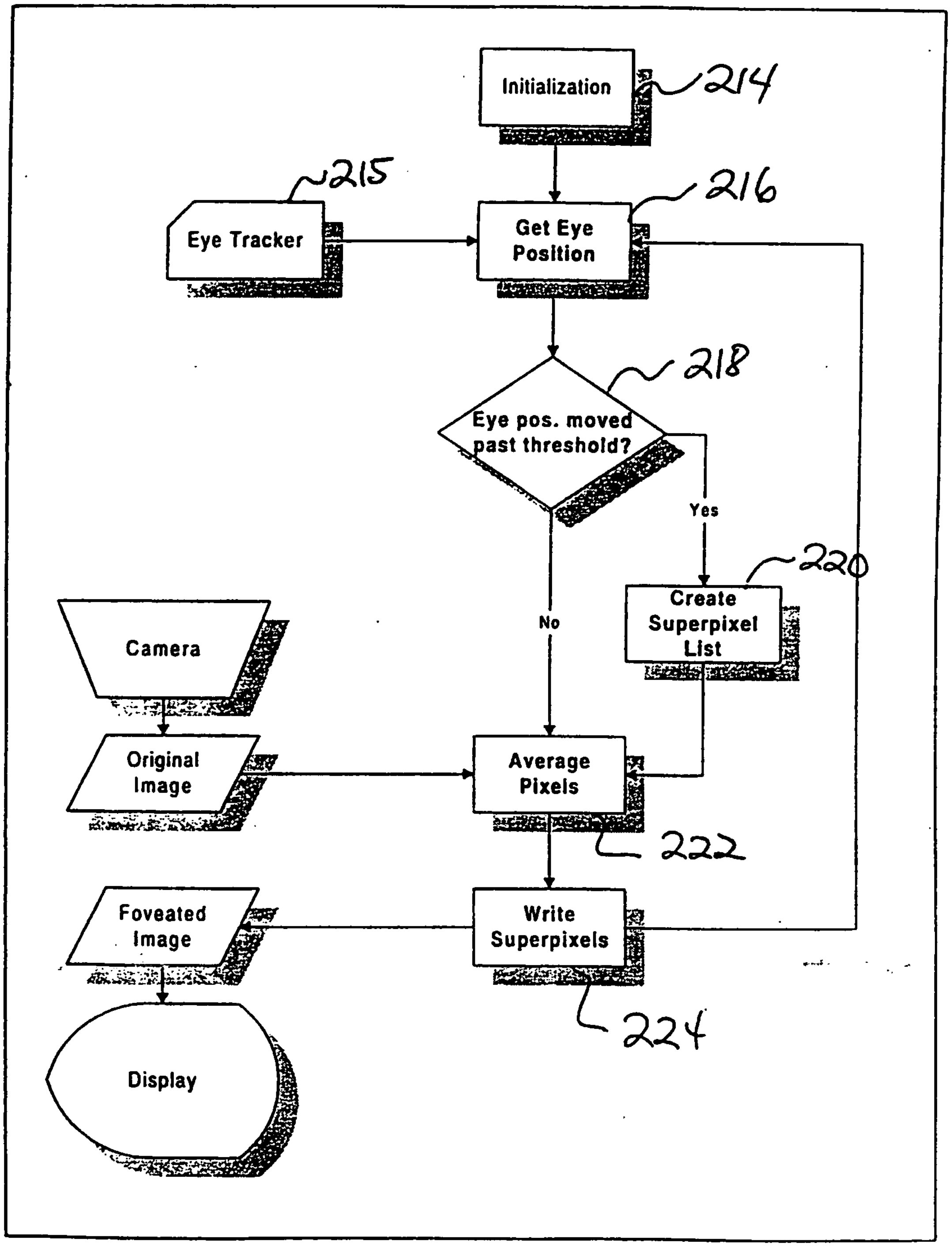

200
Initialize parameters and Calibrate
202
Calculate Resolution Grid
204
Determine eye position
206
Compute offset from last eye position
208
Has eye moved more than threshold amount ?
NO
YES
Add off-set to coordinates
209
Apply averaging algorithm to the image
210
Display SuperPixels to the screen
212

FIG. 10


Initialization
214
Eye Tracker
215
Get Eye Position
216
Eye pos. moved past threshold?
218
Yes
No
Create Superpixel List
220
Camera
Original Image
Average Pixels
222
Foveated Image
Write Superpixels
224
Display

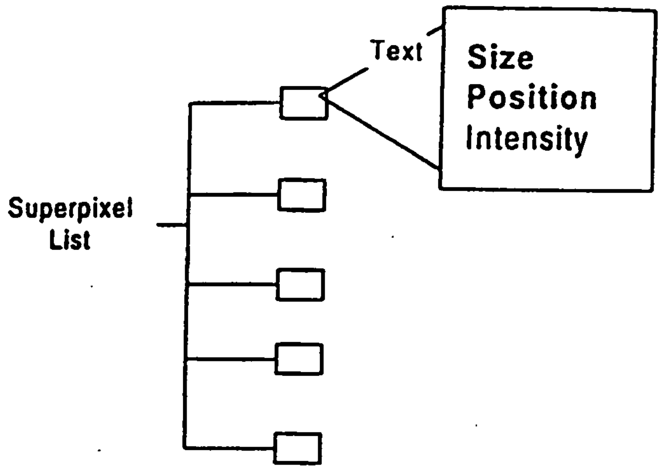


FIG. 11A

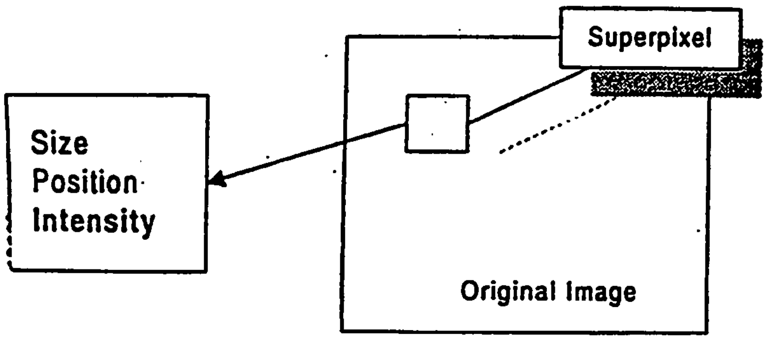


FIG. 11B

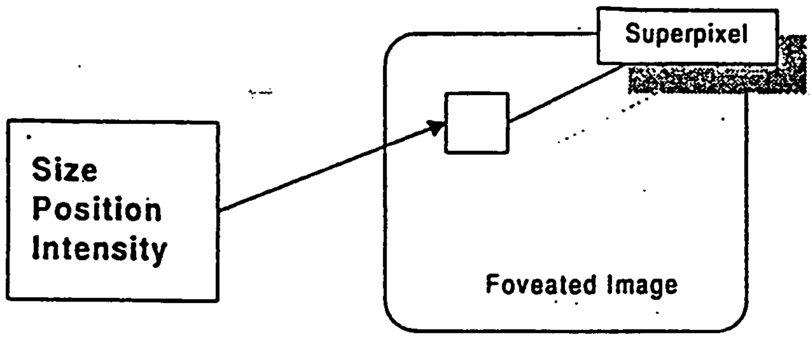


FIG. 11C

FIG. 12

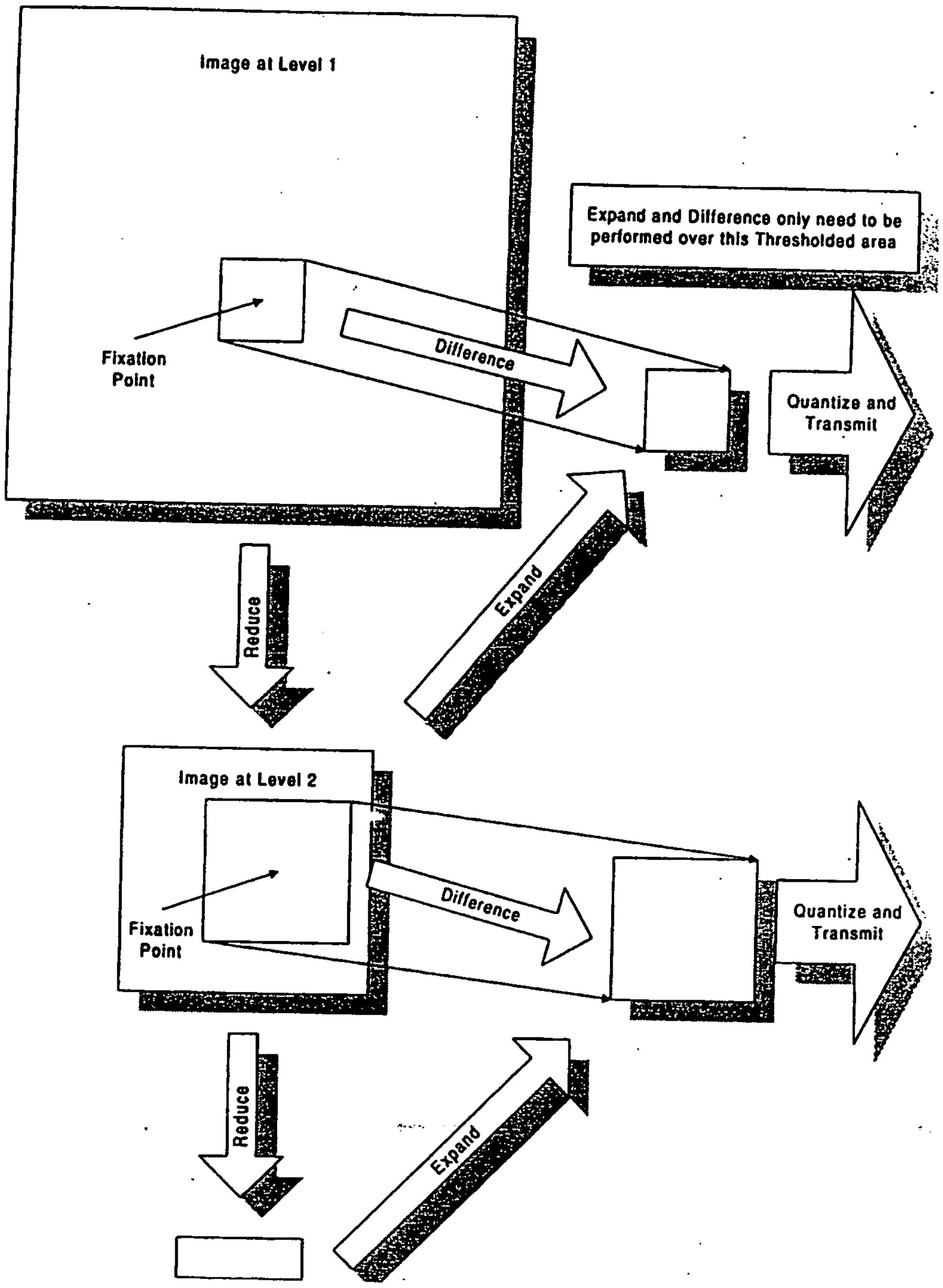


Image at Level 1
Fixation Point
Difference
Expand and Difference only need to be performed over this Thresholded area
Quantize and Transmit
Reduce
Expand
Image at Level 2
Fixation Point
Difference
Quantize and Transmit
Reduce
Expand

FIG. 13

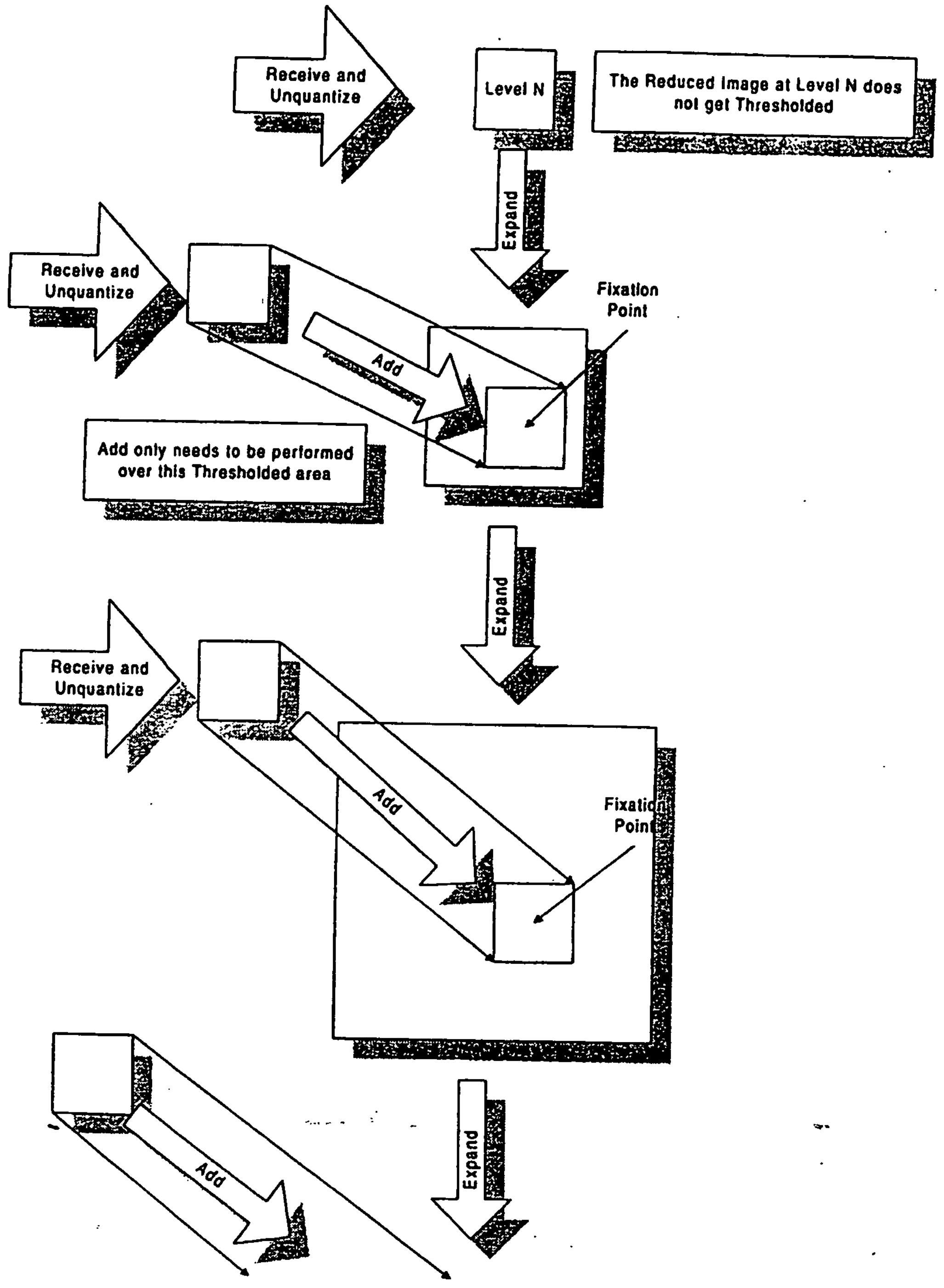

Receive and Unquantize
Level N
The Reduced Image at Level N does not get Thresholded
Expand
Receive and Unquantize
Fixation Point
Add
Add only needs to be performed over this Thresholded area
Expand
Receive and Unquantize
Add
Fixation Point
Add
Expand

FIG. 14

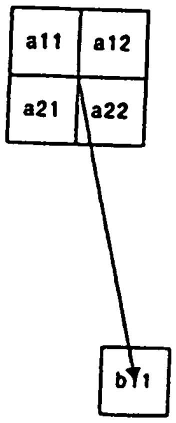


FIG. 15

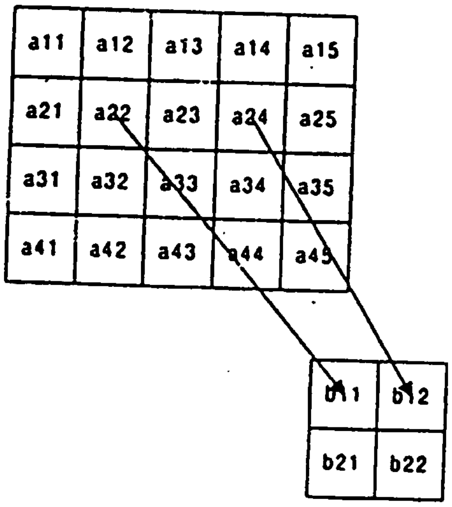

a11
a12
a13
a14
a15
a21
a22
a23
a24
a25
a31
a32
a33
a34
a35
a41
a42
a43
a44
a45
b11
b12
b21
b22

FIG. 16

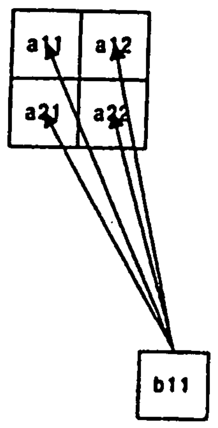


FIG. 17

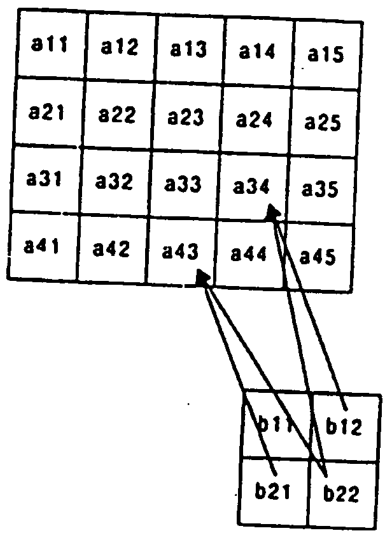

a11
a12
a13
a14
a15
a21
a22
a23
a24
a25
a31
a32
a33
a34
a35
a41
a42
a43
a44
a45
b11
b12
b21
b22

FIG. 18

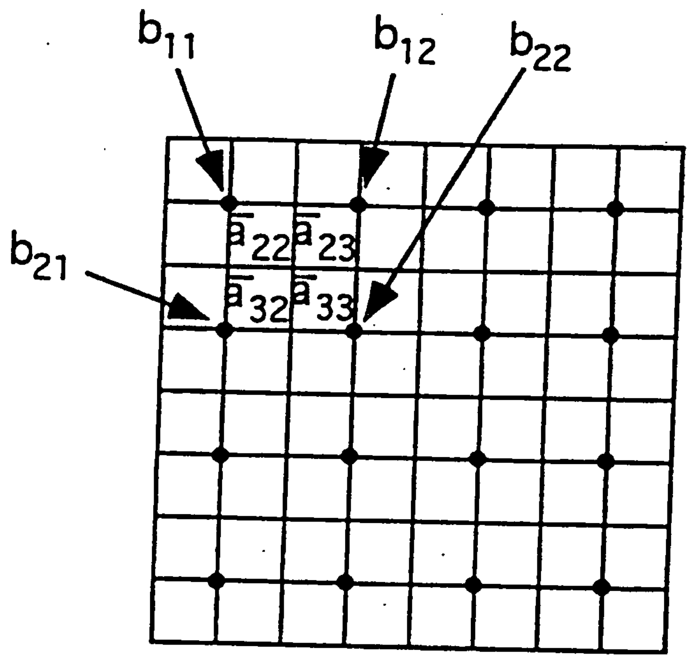
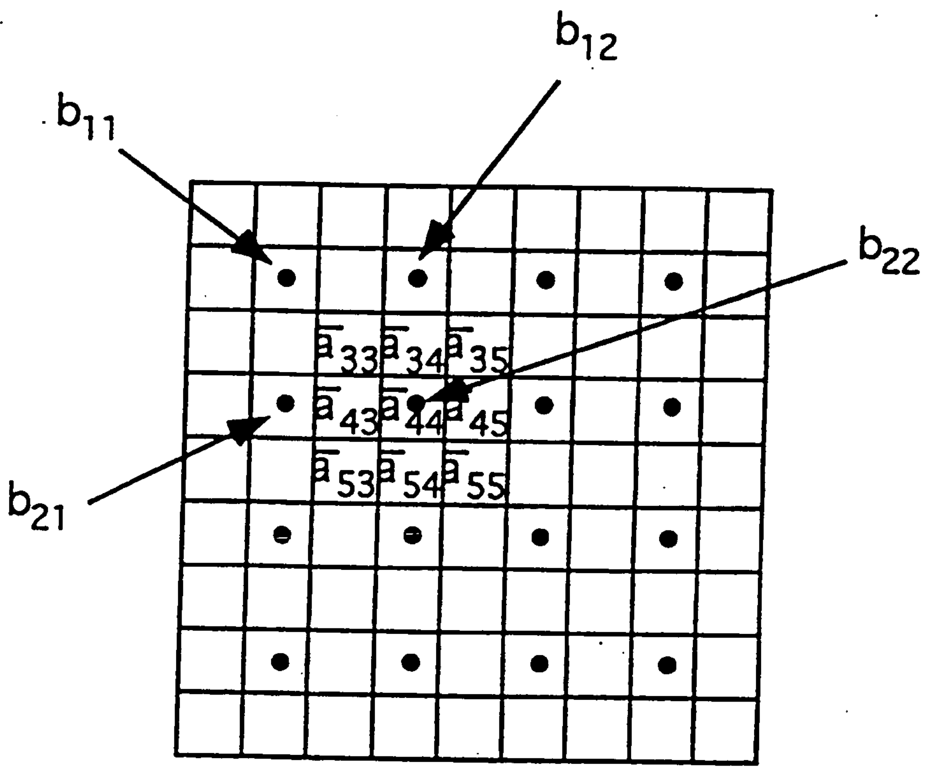


FIG. 19


b_{11}
b_{12}
b_{21}
b_{22}
$\bar{a}_{33}$
$\bar{a}_{34}$
$\bar{a}_{35}$
$\bar{a}_{43}$
$\bar{a}_{44}$
$\bar{a}_{45}$
$\bar{a}_{53}$
$\bar{a}_{54}$
$\bar{a}_{55}$